EFFICIENT RUNTIMES FOR GRADUAL TYPING

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

FABIAN MUEHLBOECK (MÜHLBÖCK)

December 2019

EFFICIENT RUNTIMES FOR GRADUAL TYPING

FABIAN MUEHLBOECK

Cornell University, 2019

This dissertation concerns the design and implementation of programming languages featuring gradual typing—which is the idea that some parts of a program may be type-checked dynamically while others are type-checked statically. This lets programmers trade off between the costs and benefits of using static type-checking for each individual part of their program as needed, and even eventually change their decisions about those trade-offs.

Designing gradually typed languages has its own trade-offs: existing gradually typed languages had to essentially decide between being efficient versus behaving in expected and safe ways. Since many of those languages were just gradually typed variants of existing languages, those trade-offs were largely forced by the original language design.

Here, we look at the design questions around gradual typing in an unconstrained scenario—what if we design a new language featuring gradual typing from the ground up? In particular, we explore these questions for nominal object-oriented programming languages. Designing a new language from the ground up lets us *co-design* the features of the language and its implementation. Accordingly, in this dissertation, we tackle a variety of design questions of particular importance to gradual typing, such as decidable subtyping, as well as questions of

implementation, most importantly efficient casting techniques, which we evaluate using benchmarks from the literature on efficiency in gradual typing.

The results presented here show that when gradual typing is co-designed with the rest of the type system and with an eye towards efficiency, it is possible to obtain both the desired formal properties proposed so far for gradual type systems and very low overheads due to gradual typing. This points the way towards a new generation of programming languages that can be used to seamlessly transition between personal scripting or rapid prototyping and large-scale software engineering.

BIOGRAPHICAL SKETCH

Fabian Muehlboeck was born in Wels, Austria. He obtained a BSc in Software & Information Engineering from TU Wien (Vienna, Austria) and an MS in Computer Science from Northeastern University (Boston, MA, United States), where he was a Fulbright Exchange Student. After obtaining his Ph.D. in Computer Science from Cornell, he will be a postdoctoral researcher at IST Austria.

ACKNOWLEDGMENTS

This dissertation is the culmination of six years in Cornell's PhD program and all the things that happened before then in order to get there. There are so many people to thank for providing all kinds of help and support on the way. I owe a lot of gratitude to so many friends, colleagues, and mentors—at Cornell, at Google, at Northeastern, at PBS Logitek, and at TU Wien. There are far too many of them to name them all here, but a few people and institutions stand out even more than others:

First and foremost, thanks to my advisor, Ross Tate, who not only shared my vision of programming language design research that is mindful of and accessible to programming language designers in industry, but also had the expertise and patience to guide me along this path.

Thanks to my other committee members, Dexter Kozen and Richard Miller, who greatly supported me and helped me learn a ton of interesting things that would have otherwise not been immediately on my path.

Thanks to other co-authors, Ben Greenman and Cosmo Viola, whose skill and dedication was inspiring and who were a pleasure to work with.

Thanks to Kevin Bierhoff, who gave me a great opportunity to learn about work in programming languages in industry and in particular to get lots of practical experience with intermediate code.

Thanks to Matthew Milano, who was not just an amazing roommate and friend, but also an excellent rubber duck for discussing research, and who answered an insanely high amount of C++ questions.

Thanks to Becky Stewart, who is an amazing Assistant Director of the PhD program and has skillfully shepherded many cohorts of PhD students through the program, including me.

Thanks to the Cornell CS department as a whole, its faculty, staff, and students; I'll be forever grateful that I could spend some time with you. There were many people that helped me get there; thanks in particular to my Master's advisor Mitchell Wand and second Master's Thesis reader Amal Ahmed at Northeastern University, and to the PhD students in the Programming Research Lab at Northeastern, in particular Paul Stansifer and Dionna Amalie Glaze, who helped tremendously with my Master's Thesis.

In turn, I only got to study at Northeastern because of the Fulbright Program—lots of thanks to the citizens of Austria and the United States, and to the Austrian Fulbright commission, for giving me this opportunity.

Thanks to Franz Puntigam at TU Wien, who showed me how interesting the field of programming languages is.

Thanks to the National Science Foundation, whose CAREER grant to my advisor[1] funded me for most of my PhD.

Lastly, thanks to my parents, Manuela and Wolfram, and the rest of my amazing family, whose love and support I have been able to count on since even before I could count.

1 This material is based upon work supported by the NSF under grant CCF-1350182. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the NSF. The same holds for the Fulbright program and any other people and institutions mentioned in here except for sometimes my collaborators.

The work in this dissertation itself would be a lot worse or non-existent were it not for the helpful feedback of my thesis committee, numerous reviewers at various instances of OOPSLA, POPL, and PLDI, various members of the gradual typing research community, and the programming language research groups at Cornell, Northeastern, TU Wien, and IST Austria. Thanks in particular to Julia Belyakova, Avik Chaudhuri, Benjamin Chung, Jonathan DiLorenzo, Molly Feldman, Matthias Felleisen, Ronald Garcia, Ben Greenman, Tom Henzinger, Andrew Hirsch, Basil Hosmer, Andrew Kent, Dexter Kozen, Stephen Longfield, Tom Magrino, Richard Miller, Greg Morrisett, Andrew Myers, Francesco Zappa Nardelli, Max New, Artem Pelenitsyn, Benjamin Pierce, Franz Puntigam, Gregor Richards, Adrian Sampson, Isaac Sheff, Jeremy Siek, Éric Tanter, Ross Tate, Ewan Tempero, Laure Thompson, Sam Tobin-Hochstadt, Jan Vitek, Michael Vitousek, IFIP WG 2.16, the Ceylon Team, and the Kotlin Team.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

0 INTRODUCTION

0.1 GRADUAL TYPING

"[...] languages that literally provide static and dynamic typing in the same program, with the programmer controlling the degree of static checking by annotating function parameters with types, or not. We use the term ***gradual typing*** for type systems that provide this capability." *Siek and Taha [2006]*

Gradual typing is the relatively recent idea that programming languages need not exclusively follow the *static* or the *dynamic typing* discipline [Gronski et al., 2006; Matthews and Findler, 2007; Siek and Taha, 2006; Tobin-Hochstadt and Felleisen, 2008]. Rather, in gradually typed languages, programmers can trade off the costs and benefits of static and dynamic *type-checking* differently in different parts of their programs, and ideally also change their opinion on those trade-offs later. Usually, the programmer indicates their choice of static vs. dynamic type-checking by whether or not they write down *type annotations*—having annotations means that this part of the program should be statically type-checked. For example, in Figure 0.1, the version of `add` on the left would be dynamically type-checked, and the version of `add` on the right would be statically type-checked.

```
function add(x, y)
{
  return x+y;
}
```

```
function add(int x, int y) : int
{
  return x+y;
}
```

Figure 0.1: A function without (left) and with (right) type annotations

What are the important differences between the two? On the one hand, the statically typed version on the right is now already *documented* in a way that both humans and computers can understand: the function takes two integer values and produces an integer value. From this, a programmer can already infer a lot about how they can use the function, and a computer can use this both to check that the supposed input-output behavior holds (in this case by checking that in turn, the basic operation "+" produces an integer when given two integer arguments), and to optimize the code (in this case by selecting the plain integer addition operation implemented in the computer's hardware).

The dynamically typed version on the left has none of these advantages; in particular, as "+" may also work on other kinds of data, like floating point numbers or strings, the function cannot be optimized in the same way as the statically typed version—the decision about what exactly to do has to be deferred to run time. On the other hand, this also means that the dynamically checked function is more flexible—it will accept all kinds of data and try to run "+" on them, and that might just work, like for the aforementioned floating point numbers and strings. If, during the run of the program, it turns out that `add` was given two arguments that "+" cannot process, then the program crashes and tells the programmer that there was a problem in the function `add`.

In contrast, static type checking gives the programmer a guarantee that the program will never crash in this way, because if `add` is only ever given integers, we know that "+" can process them, and the static type checker will in turn also check that everything that is given to `add` can be processed by it (i.e. is an integer), and so on. This is core to how static type checking works: every single part of the program is checked to make sure

that, whenever it interacts with other parts of the program, it respects the annotations of those other parts. When every part respects the annotations of each other part, we can guarantee that an important class of bugs does not occur when running the program, namely passing the wrong kind of data to an operation.

0.2 GRADUAL TYPING IN INDUSTRY

It turns out that the requirement that one can guarantee that all parts of a program work together perfectly is rather restrictive, in particular for smaller-scale programming tasks—in order to satisfy it, one might have to write a bunch of code to handle every possible corner case in one's program. This is particularly cumbersome for people who are just learning how to program, or for code where it is fine to fail if anything falls outside of the expected parameters, such as scripts, prototypes, or code for interfacing with databases or other programs.

It is thus no surprise that a large amount of software written today is written in dynamically typed languages, where one can indeed just write the code that is supposed to actually run and not care about any special cases. JavaScript [ECMA, 2019; Guha, Saftoiu, and Krishnamurthi, 2010] is the main language of the world wide web, Python [Politz et al., 2013; Rossum, 1995], R [Ihaka and Gentleman, 1996], and Matlab [The MathWorks, Inc., 2019] have widespread use in science and engineering, and many website backends are still written in PHP [The PHP Group, 2019]. This is the case despite the downsides of those languages, of which there are two major ones.

First, dynamically typed languages are usually interpreted and orders of magnitude slower than their typed and compiled counterparts. The way Python, R, and Matlab are thus applied in the sciences is mostly as a scripting interface for powerful and highly optimized libraries written in languages like C and C++. These scripts are written in such a way that offloads most of the work to those libraries; however, even the more optimized libraries may take hours or days to process the large quantities of data many fields have to deal with today. Thus, a sad experience many scientific programmers endure is that after a long time of processing, they discover that a small bug—and oftentimes one that a type checker would have found—made the program crash and potentially obliterated all the results of all the hard work the program just did. Moreover, many users of those high-performance libraries do not have any expertise in the language the libraries were written in and so cannot even make slight changes to their interfaces or develop small variants—there is a hard barrier between the scripting language they work in and the libraries that do the computationally expensive work.

Second, the checks and documentation provided by type annotations and static type checking are particularly useful for large software projects where sometimes hundreds of engineers work on the same code base. Type annotations help programmers to pass the right kinds of data to code written by other programmers, and static type checking notifies them immediately if they get it wrong. For example, Facebook's codebase was originally in PHP and JavaScript, but as the codebase grew larger and larger, they eventually developed Hack [Facebook, 2016], which is essentially a gradually typed version of PHP, and Flow [Facebook, 2014], a type checker for a gradually typed version of JavaScript. Flow is not the

Part of the Hack announcement: "Traditionally, dynamically typed languages allow for rapid development but sacrifice the ability to catch errors early and introspect code quickly, particularly on larger codebases. Conversely, statically typed languages provide more of a safety net, but often at the cost of quick iteration."—*Verlaguet and Menghrajani [2014]*

only gradually typed version of JavaScript—TypeScript [Microsoft, 2012] also enjoys widespread use. What Hack, Flow, and TypeScript have in common is that they are gradually typed in the sense that one can mix dynamically and statically type-checked code, which was necessary to be able to interact with the large amount of existing code and to be able to smoothly transition to a more typed code base. However, they are all also unsound.

0.3 SOUND GRADUAL TYPING

Unsoundness means that the type annotations only serve as (unchecked) documentation; the best a type checker can do with them is find those annotations that are clearly inconsistent with other annotations. Recall that static type-checking depends on the global assumption that every single part of the program is checked to play well with all the others. When even a tiny part of the program is not subject to static type-checking, it invalidates this assumption and therefore all guarantees that static type-checking is supposed to provide. The lack of these guarantees also precludes using the type information to optimize the code more, as it would be highly dangerous to rely on faulty type information.

In contrast to the unsound gradual typing employed by the industry languages mentioned above, there are also a number of sound gradually typed languages mostly developed in academia, among them Typed Racket [Tobin-Hochstadt and Felleisen, 2008], Reticulated Python [Vitousek, Kent, et al., 2014], GradualTalk [Allende, Callaú, et al., 2014], and Safe TypeScript [Swamy et al., 2014]. To avoid the violations of the assump-

tions the static type checker makes, these languages insert run-time checks to ensure that those assumptions always hold. What that means, for example, is that when the "typed" version of `add` is called from "untyped" code, the arguments are checked at run time to ensure that they are the integers that the typed version of `add` expects—if not, an error can be raised right there, before we enter a region of the program that is supposed to be free of such errors. There is just one problem with these languages: so far, all of them either experience significant overheads because of those run-time checks in programs where typed and untyped code are mixed, or have extremely inefficient code at either the fully typed or fully untyped end of the spectrum. Until Takikawa et al. [2016] proposed a systematic way of measuring the overhead of gradual typing and found huge overheads for Typed Racket, this had not received too much attention, but since then there has been a flurry of work to address the efficiency problem [Bauman et al., 2017; Feltey et al., 2018; Greenman and Felleisen, 2018; Greenman and Migeed, 2018; Kuhlenschmidt, Almahallawi, and Siek, 2019; Richards, Arteca, and Turcotte, 2017; Vitousek, Swords, and Siek, 2017], including work presented in this dissertation, particularly in Part II.

Technically the distinction is between "statically type-checked" and "dynamically type-checked" code, but in the rest of this dissertation we will use the more colloquial terms "typed" and "untyped".

There is yet another kind of gradually typed language out there: C# [Bierman, Meijer, and Torgersen, 2010]. Performance-wise, C# does not really suffer from huge overheads due to gradual typing, mostly because its implementation of dynamic type-checking is quite slow (for numbers, see Section 5.8). The problem for C# is that its type system was designed for static type-checking, and there are a number of problematic interactions between gradual typing and other type-system features of C#. In short, C# grossly violates a property called the *Gradual Guarantee* [Siek, Vitousek, Cimini, and Boyland, 2015], which roughly states that adding correct type

annotations to a program should not change the program's behavior. The problem is that in C#, types affect the semantics of a program, and so without type annotations it is not necessarily always clear what the semantics of a program should be if there could be multiple different valid annotations.

One thing to note about the existing gradually typed languages mentioned so far is that almost all of them are based on existing programming languages that already had large existing code bases, compilers, runtimes, and other infrastructure. This greatly reduces the room to maneuver when adding new features, such as gradual typing, which has ramifications for just about every part of language design. The goal of the work presented in this dissertation, on the other hand, is to explore the possibilities if one were to design a new programming language: we can select and design the type-system features in such a way that they both interact well with gradual typing and can be implemented efficiently.

0.4 USE CASES FOR GRADUAL TYPING

When designing a programming language feature, a central question should be what it will be used for. A gradually typed language will be used to write both typed and untyped code, where programmers trade off the advantages and disadvantages of each style as already discussed. Not only will code be written in one of those styles, it may also be transitioned from one to the other (usually from untyped to typed) at a later point in the course of maintaining and extending the code base.

Not all programmers will do all of these things; yet gradual typing may still come in handy, as whatever style they choose, libraries that they use might be written in the opposite style. In the following, we give an overview of concrete use cases one might envision for gradual typing.

"Most scripting languages are untyped and have a flexible semantics that makes programs concise. Many programmers find these attributes appealing and use scripting languages for these reasons. Programmers are also beginning to notice, however, that untyped scripts are difficult to maintain over the long run."—*Tobin-Hochstadt and Felleisen [2008]*

1. One of the first use cases of research in gradual typing was adding types to untyped languages [Allende, Callaú, et al., 2014; Facebook, 2014, 2016; Microsoft, 2012; Swamy et al., 2014; Tobin-Hochstadt and Felleisen, 2008; Vitousek, Kent, et al., 2014]. In this use case, programmers may write both typed and untyped code, interact with libraries both styles, and transition code between one and the other.

2. In programming education, it is useful for students to be able to experiment. Experimental code is not necessarily complete; having to satisfy a type checker that a student does not yet really understand is an obstacle to building the algorithmic intuition they are supposed to build early on. On the other hand, it is useful to interact with (typed) standard libraries that are documented with types to state clearly how they are supposed to be used and to give informative error messages when they are used in the wrong way. Thus, gradual typing can be useful in this setting, even though students themselves might (at first) only write untyped code.

3. Scripts are simple programs that are written with a particular task in mind. They are usually written by a single programmer with a lot of specific assumptions about the state of the system and the input data that a type checker has no way of knowing, and it is completely acceptable for the script to crash if those assumptions are violated. Many of the dynamically typed languages mentioned above are

also called scripting languages because they are well-suited for this use-case. Like in education, the usefulness of dynamic type-checking only applies to the actual script; having the core libraries and any other libraries that are used be typed has the same advantages here as above. In addition, sometimes scripts grow beyond the narrow task they were originally written for, in which case the ability to transition them to typed code is immensely useful.

4. Even relatively simple programs often need to interact with entities outside of the program, for example the file system, other programs like databases, or high-performance libraries. The interfaces to those entities do not generally perfectly match the type system of one's programming language, and are usually written as a best possible fit that works if used correctly. Gradual typing allows those interfaces to be typed where possible, and untyped where not, while not forcing the programmer to write extra code and annotations to work around the type checker. In such situations programmers that normally write typed code would benefit from being able to write untyped code in some places.

5. Lastly, and separately from adding types to untyped languages, software developers can use gradual typing to greatly enhance their experience in the software development cycle. Gradual Typing enables them to rapidly build prototypes in untyped code—if those prototypes are successful and their code stabilizes, programmers can then add type annotations to obtain the benefits of static type-checking, while still using untyped code for experimental new additions to the code base. This is a benefit of any gradually typed language that

satisfies the major formal properties that researchers have come up with so far.

Of these use cases, we already mentioned that the first (adding types to untyped languages) has received ample attention, but also severely constrains the design choices one can make both in terms of the type system and the implementation of gradual typing. It is therefore not the focus of this dissertation, while the other four are the underlying motivation for the work presented here.

0.5 A ROADMAP FOR PRACTICAL, SOUND, AND EFFICIENT GRADUAL TYPING

The goal of this dissertation is to inform the design of gradually typed object-oriented programming languages that apply to the latter four use cases discussed above. We aim for these languages to be efficient and "well-behaved", that is to say:

1. Static type checking is sound: the static type system does provide actual guarantees about the program that can be used to reason about and optimize it
2. Type-system features interact well with each other and in particular gradual typing
3. It is possible for untyped code to be transitioned to typed code with little effort, usually just by adding the necessary annotations

To achieve those goals, we will leverage the fact that we are free to pick and design the language and its type-system features such that they are compatible with gradual typing *and* can be implemented efficiently.

0.5.1 *Nominality and Typed Libraries*

The first design choice is that the core of the language is a nominal, class-based type system like the ones found in Java and C#, and that, like in those languages, the core libraries of the language are going to be typed. Nominality, as we will see, enables us to have very efficient run-time type checks, which we need for soundness. The design choice also allows us to draw from the feature set of major industrial programming languages. Finally, it is compatible with all the four latter use cases discussed above—in each of them, typed core libraries will enhance the experience of programmers by providing better documentation, checks and guarantees ahead of running the program, informative error messages when those checks fail, and much more efficient code; the downside of typed code is usually more effort to develop those libraries, which is easily justified for the core libraries of a language.

Nominal *type systems are those where types are identified by their names, not their structure. This makes it such that types can be represented very compactly, and type-checking algorithms do not have to recurse through the structure of the type, but through relations between the type names that the programmer specified, if any.*

0.5.2 *Milestones*

To make a gradually typed, nominal, object-oriented language practical, it should have type-system features similar to those found in such languages today. That is, it should feature things like inheritance, generics and

anonymous functions ("lambdas"), and possibly some form of overloading. None of these are trivial to include.

The form of those features in current major programming languages was designed for static type checking: checks happen at compile time, and so do some type-based decisions that have run-time consequences—type annotations often affect the semantics of a program, for example in overloading or generic type-argument inference. With (sound) gradual typing, some checks now may have to be deferred to run time, and some type annotations may be changed or not exist at all, which affects any type-based compile-time decisions—in order for gradual typing to support transitioning from untyped to typed code, adding or changing a type annotation should not change "what the program does".

Our goal for any type checks that are deferred to run-time is pretty clear: they should finish quickly, and, in particular, they should finish at all. The latter goal is not a given, either: Grigore [2017] showed Java's type system to be undecidable, because of subtyping with generics. However, while *compile*-time crashes in rare cases may be acceptable, unpredictable crashes at *run* time are not. This means that in order to have any hope of making gradual typing work reasonably with a nominal object-oriented language with generics, we need subtyping to be decidable for such languages.

Milestone 1 (Decidability)**.** *Subtyping and type checking must be decidable.*

With practical ways of making type checking decidable, we can work on adapting type-system features for efficient gradual typing. Besides efficiency, it is important to satisfy several theoretical properties, the most important of which is the *Gradual Guarantee* [Siek, Vitousek, Cimini, and Boyland, 2015]. It roughly says that different but valid type annotations

for a program should not change the semantics of the program—this is important to allow programmers to transition untyped to typed code without suddenly changing what the program does. With a basic set of type-system features that work with gradual typing in hand, we can show that a nominal object-oriented language with these features can have efficient run-time type checks.

Milestone 2 (Efficiency)**.** *Run-time type checking must be efficient.*

Once we have a foundation of an efficient gradually typed language, we can extend it to have more interesting features. Important features for modern object-oriented programming languages are generics and lambdas, both of which usually use some form of local type inference to be practical. For lambdas, that is particularly problematic with respect to gradual typing, as their type inference depends on type annotations in their context to determine the types of arguments, based on which the return type of the lambda is calculated. Gradual typing means that the relevant type annotations in the context or anywhere in the code of the lambda may be missing, such that its return type is now unknown. However, other typed code somewhere in the program may still expect the function to have a particular type, such as `Int` $\rightarrow$ `Int`. For an arbitrary untyped function, it is impossible to immediately prove that it conforms to this type. A large amount of work in the area of gradual typing has been spent on finding and classifying ways of dealing with this problem [Chung et al., 2018; Greenman and Felleisen, 2018; Greenman and Migeed, 2018; Siek, Vitousek, Cimini, Tobin-Hochstadt, et al., 2015; Tobin-Hochstadt and Felleisen, 2008; Vitousek, Kent, et al., 2014]. We similarly need to find one

that can be adapted to let lambdas, and ideally even structural records fit into our otherwise nominal type system.

Milestone 3 (Structural Values)**.** *Support for structural values must integrate with the nominal type system and still be efficient.*

Next, the usability of generics is greatly improved by generic type-argument inference, where generic type parameters are inferred from the arguments given to a function. The algorithms used for this in major industry languages right now are rather ad hoc, incomplete, and prone to change the semantics of programs with slight changes in the type annotations or just patches to the compiler [Smith and Cartwright, 2008]. To support gradual typing, type-argument inference needs to be more stable in the face of changing precision of type annotations, and also ideally be able to deal with structural higher-order values like lambdas.

Milestone 4 (Practical Generics)**.** *Generics need to support type-argument inference that is compatible with gradual typing, and ideally also with structural values.*

For each of these milestones, the work presented in this dissertation advances the current state of the art. Of course, there is much more to do even after those milestones are completed—some important aspects left to future work are discussed at the end in Section 9.1.

0.6 CONTRIBUTIONS IN THIS DISSERTATION

The concrete major contributions discussed in this dissertation are:

1. A practical restriction to inheritance relations that makes subtying in nominal object-oriented languages with variant generics decidable,

using a straightforward subtyping algorithm (Chapter 2, adapted from [Greenman, Muehlboeck, and Tate, 2014]).

2. A framework for extending type systems with decidable subtyping algorithms to also feature union and intersection types(Chapter 3, adapted from [Muehlboeck and Tate, 2018b]).

3. Experimental results showing that sound gradual typing can be implemented efficiently, at least in a purely nominal type system (Chapter 5, adapted from [Muehlboeck and Tate, 2017b]).

4. Experimental results indicating that purely nominal type systems can be extended with lambdas and records in a way that can still perform efficiently (Chapter 6).

5. New ways of formalizing gradual typing and design insights for various type-system features interacting with gradual typing (Chapters 5, 6, and 8).

6. A demonstration that not all languages can be gradualized due to the influence types often have on semantics in typed industry languages (Chapter 7)."

7. A formalization for when generic type arguments can be inferred in a principled and decidable manner, and in a way that avoids known incompatibilities with gradual typing (Chapter 8).

All in all, this dissertation will hopefully be useful to whoever works on designing new, gradually typed, object-oriented languages or wants to add to this line of research.

Part I

DECIDABLE SUBTYPING

1

OVERVIEW

Decidability is often a property that is viewed as nice to have, but not necessarily essential. In fact, many existing statically typed programming languages that are in widespread use have type systems that are undecidable, among them Java [Grigore, 2017], Scala [Dürig, 2010], Haskell [McBride, 2002], and, because of their basis in Java, Kotlin [JetBrains, 2019] and Ceylon [King, 2013]. Moreover, System $F_{\leq}$, the λ-calculus with subtyping and bounded second-order polymorphism, is undecidable [Pierce, 1992]. For some languages, like C# [Kennedy and Pierce, 2007] and Julia [Zappa Nardelli et al., 2018], it is not yet known whether their type system is decidable (though at least for C# it is very likely). For fully statically typed languages, one may see the type system as "decidable enough in practice", meaning that programmers will most likely not encounter the undecidable corner cases of the type system in actual programs that they write. Even if they do, the program simply does not compile, as the type checker will either crash or signal some other error. While this scenario is annoying, in particular because crashing compilers usually do not give good error messages that a programmer can use to fix the problem, no end user of a compiled program will ever be affected by this undecidability. The problem becomes a lot more acute in the setting of sound gradual typing, as it relies on run-time type checks, which, if they are not decidable, may unpredictably crash a program while it is running. So, in addition to the

side-benefits of enabling better tooling for programmers in statically (and gradually) typed languages, having a decidable type system is a practical necessity for soundly gradually typed languages.

The chapters in this part of the dissertation cover two important results in decidable subtyping: first, in Chapter 2, that there is a practical way of restricting inheritance definitions and type argument constraints in class-based object-oriented languages like Java and C# that makes subtyping decidable, even using a straightforward specification and algorithm. We call the underlying principle *Material-Shape separation*, which in short divides type declarations into those that are used to describe other types in recursive inheritance declarations or type argument constraints (Shapes), and those that are used to describe data that is passed around in the program (Materials). This result is a key foundation for the rest of the work discussed in this dissertation. Every other chapter assumes Material-Shape separation to guarantee decidability of subtyping.

The second result discussed in this part (Chapter 3) covers the decidability of more advanced type-system features, in particular in connection with union- and intersection types. Union- and intersection types are quite old and powerful concepts, but have until recently not shown up in many actual programming languages, much less both of them at once. One reason for that is that the typing and subtyping algorithms that cover interactions of union and intersection types with each other and other type-system features are non-trivial, leading to a large number of works on particular decidable subtyping algorithms for particular type systems involving union and intersection types [Ancona and Corradi, 2016; Castagna and Xu, 2011; Dardha, Gorla, and Varacca, 2013; Frisch, Castagna, and Véronique Benzaken, 2002, 2008; Gochet, Gribomont, and Rossetto, 2005;

Hosoya and Pierce, 2003; Viganò, 2000]. Chapter 3 describes a machine-verified framework that allows decidable type systems to add union and intersection types such that the resulting type system is decidable and features distributivity between unions and intersections. Futhermore, the framework allows its users to reason about the interactions of union and intersection types with other type-system features to provide even more powerful reasoning (such as disjointness and distributivity over generics and function types).

Besides becoming a more and more popular feature in programming languages in general, union and intersection types can be useful for generic type-argument inference, as it relies on being able to compute meets and joins, which unions and intersections trivially represent. Chapter 8 discusses a type system where this is exploited, and the results in Chapter 3 show that such a type system can have several advanced features while still retaining decidability.

2

DECIDABLE SUBTYPING WITH VARIANT GENERICS

This chapter is based on a paper presented at PLDI 2014: *Getting F-Bounded Polymorphism into Shape* [Greenman, Muehlboeck, and Tate, 2014].

2.1 INTRODUCTION

Generics were a long-awaited addition to Java and C#. They finally gave industry developers access to the benefits of parametric polymorphism. But polymorphism was not originally designed for object-oriented languages, rather it was tried and tested primarily in functional languages [Milner, 1978]. There is one fundamental difference between typical instances of these language classes: subtyping. The design and algorithms for polymorphism were centered around unification [Baader et al., 2001], a technique that only works smoothly in type systems without subtyping. Yet subtyping is a key part of Java, C#, Scala, and numerous other object-oriented languages, and the question of how to combine polymorphism and subtyping needs a solution that is capable of expressing the various idioms used in practice while still providing sound and complete algorithms for type checking. Here we provide the foundations of such a solution, one based on reshaping F-bounded polymorphism [Canning et al., 1989] to properly match how it is used in practice.

In the rest of this chapter, we refer to parametric polymorphism simply as polymorphism, excluding other meanings such as subtype polymorphism.

Plain bounded polymorphism is the ability to specify the range of types a type variable can represent. Typically this is done with an upper bound, i.e. a constraint indicating what classes/interfaces instantiations of a type variable must implement. This allows the programmer to guarantee the presence of various methods, such as requiring a type variable to extend `Formattable` so that the programmer can safely use the `format` method. Thus bounded polymorphism enables programmers to impose the same requirements and guarantees on type arguments that they can impose on function parameters and returns.

F-bounded polymorphism is the ability to constrain a type variable by a type expressed in terms of the type variable itself [Canning et al., 1989]. In other words, F-bounded polymorphism is the ability to use recursive constraints. This subtle addition significantly increases the power of type-variable constraints. In particular, F-bounded polymorphism addresses the issue of *binary methods*, the pattern that operations such as comparison and addition need both arguments to have the same type. With F-bounded polymorphism, one can require a type parameter `T` to extend `Comparable`⟨`T`⟩, where `Comparable`⟨`T`⟩ has a comparison method that only accepts arguments of type `T`. This way types such as `Integer` and `String` can be compared to themselves but not to other types that happen to also have a comparison method. Java's equality design does not adopt this paradigm, instead declaring equality to exist between all objects. Consequently, the type checker cannot help identify cases where the wrong types of objects are being compared, and most implementations of `equals` have to first cast its parameter to the correct type.

The main drawback of F-bounded polymorphism is that it requires inheritance to be recursive. For example, the standard class `String` imple-

ments `Comparable⟨String⟩`, so the inherited type is defined in terms of the inheriting type itself. On its own, this is a simple feature, but generics typically also have some form of *variance*. That is, a `List⟨String⟩`[1] can safely be treated as a `List⟨Object⟩`, an ability that is rather useful in practice, and consequently languages such as C# and Scala provide a way for programmers to declare that `List` is *covariant* [Hejlsberg, Torgersen, et al., 2010; Odersky, Altherr, et al., 2014]. Dually, something that is comparable to arbitrary objects can safely be compared to integers, making `Comparable` *contravariant*. Unfortunately, the combination of variance and recursive inheritance greatly complicates many type-checking algorithms. Indeed, Kennedy and Pierce proved that even just subtyping is undecidable in languages supporting these two features [Kennedy and Pierce, 2007].

The key insight in this chapter is that we can recover decidability and algorithmic simplicity by restricting recursive inheritance to how it is actually used in practice. We call the classes/interfaces used for recursive inheritance, such as `Comparable`, *shapes* because they describe the higher-level shape of the type using recursive inheritance. What we recognize is that shapes are used in a very restricted fashion in practice. In particular, shapes are never used in parameter types, return types, field types, and type arguments. Instead, we call the classes/interfaces used in those locations *materials*, because they are the types actually used for material exchanges across the components of a program. Our fundamental finding is that, should one require materials and shapes to be disjoint sets, the 13.5 million lines of generic-Java code we analyzed would be unaffected except for where the analysis identified flaws in the designs. We call this observed property *Material-Shape Separation*.

1 We use `List` to represent some read-only list interface.

With this understanding of industry code, we are able to formalize a decidable type system that is backwards compatible with Java as it exists in practice. The key insight is that most algorithms need only be defined on materials, since shapes are only ever used as constraints. Because shapes encapsulate all recursive inheritance, inheritance amongst materials is well founded, so even naïve strategies are guaranteed to terminate. With this, previously open problems such as computable joins can be solved with simple, direct, and efficient machinery.

The rest of the main content of this chapter is organized as follows:

- Anecdotal evidence suggesting that Material-Shape Separation is already an unrecognized idiom (Section 2.2)
- A type-theoretic formalization of materials and shapes and Material-Shape Separation (Section 2.3)
- A large survey of industry code demonstrating the compatibility of Material-Shape Separation with practice (Section 2.4)
- Type-checking algorithms exploiting Material-Shape Separation to achieve simplicity and decidability (Section 2.5)
- Potential applications of Material-Shape Separation to open type-checking challenges and new type-system features (Section 2.6)

2.2 BACKGROUND

Polymorphism and subtyping make a powerful combination, and as such both have been widely adopted by statically-typed major industry languages. They also make for a troublesome combination, as Kennedy and

Pierce [2007] have shown that even subtyping with variant generics is undecidable without restriction. Consequently, Kennedy and Pierce provided various restrictions that ensure decidability, the most notable of which is banning expansive inheritance, which has been adopted by C# [Hejlsberg, Torgersen, et al., 2010]. But that solution requires a complicated algorithm, has poor blame properties, and does not work for more powerful systems like Java's wildcards [Torgersen et al., 2004]. Tate, Leung, and Lerner [2011] proposed an alternative restriction that guarantees decidability for wildcards and uses a more efficient algorithm, but their restriction is less accommodating of contravariance. Regardless of which one might be better, both solutions grew from algorithmic perspectives, recognizing current practice only insofar as to show backwards compatibility with existing code. Thus, their acceptability is conditioned on there not being any compelling counterexamples. However, the following interface is such a compelling counterexample to both solutions:

```
interface List⟨out E⟩
          extends Equatable⟨List⟨Equatable⟨E⟩⟩⟩ {}
```

Here the definition uses the `Equatable` interface to express type-safe equality. Equality is a binary method, and so modern object-oriented practice suggests it be formulated using F-bounded polymorphism and recursive inheritance. Thus the signature guarantees that all lists implement type-safe equality. Ideally, we would require that lists of `E` are equatable only when `E` extends `Equatable⟨E⟩`; however, most modern languages do not support such conditional inheritance, a feature we will discuss in more detail in Section 2.6.1. We bypass this limitation by making `List⟨E⟩` be equatable to lists of `Equatable⟨E⟩`, which will only exist when `E` extends `Equatable⟨E⟩`. Also, when `E` extends `Equatable⟨E⟩`, then `List⟨E⟩`

will actually be a subtype of `Equatable⟨List⟨E⟩⟩` due to the covariance of `List` (hence the **out** annotation on the type parameter `E`) and the contravariance of `Equatable` (typically expressed with an **in** annotation). Thus, `List⟨String⟩` will be equatable with itself and so can be used as, say, the type of keys for hash maps, which require equality to be defined on their keys. Consequently, this design presents a solution to the important open problem of type-safe equality on lists. In fact, we know of no alternative solution to this problem using just the expressiveness of Java's or C#'s generics.

This solution is rejected by both of the existing proposals to restrict generics for decidability. It uses expansive inheritance by having `List⟨E⟩` use `List⟨Equatable⟨E⟩⟩` in its inherited type, thereby violating Kennedy and Pierce's requirement [Kennedy and Pierce, 2007]. It also uses nested contravariance, with `Equatable` being used at a non-covariant position in the inherited type, thereby violating Tate et al.'s requirement [Tate, Leung, and Lerner, 2011]. Yet this design is being rejected for reasons that industry developers would view as purely academic. In other words, the common case is being sacrificed for the corner case. So, to design a more practical restriction to generics, one must better understand the common case.

To that end we presented this design to our industry collaborators, and to our surprise they were strongly opposed to it. Despite the lack of any type-safe alternatives, and even admitting they found it to be a clever exploitation of features, they rejected it because they felt like it violated unwritten, and up to that point unrecognized, design principles. In particular, to them `Equatable` is only meant to describe types via constraints; it is not something to be passed around in lists. Using `Equatable` as a type argument violates the accepted use of the interface. From this we devel-

oped the concept of shapes, e.g. `Equatable`, and materials, e.g. `List`, and we designed a type system and typing algorithms based on the separation of these two concepts, i.e. the *Material-Shape Separation*.

Using Material-Shape Separation, we are able to develop a simple sound and complete subtyping algorithm, one capable of incorporating type equivalence even in invariant types, a problem raised by Tate, Leung, and Lerner [2011] and not well addressed by any of the existing proposals for restricting generics. More importantly, we develop a sound and complete algorithm for computing the join of two types, a problem raised by Smith and Cartwright [2008] and also not well addressed by any of the existing proposals. We can even add higher-kinded constrained type variables and type lambdas with pointwise higher-kinded subtyping [Pierce and Steffen, 1997] and still maintain decidability of all these features. Finally, to justify that all this is indeed compatible with widespread industry practice and not just limited to our collaborators, we surveyed 13.5 million lines of open-source generic-Java code and found no violations of our design assumptions. Thus we have a decidable type system, with simple and efficient algorithms, that matches hitherto-unwritten design principles of industry practitioners.

2.3 MATERIALS AND SHAPES

In this section, we define materials and shapes in full detail. This section culminates with the formalization of Material-Shape Separation, the key observation enabling the algorithms presented in Section 2.5. But first, we must establish the formal setting we are working within.

When discussing generics, variance is an important challenge. In Section 2.2, we used declaration-site variance, which is used by C# and Scala [Hejlsberg, Torgersen, et al., 2010; Odersky, Altherr, et al., 2014]. However, here we will be using use-site variance, a simplification of Java wildcards [Gosling, Joy, Steele, and Bracha, 2005]. Tate [2013] discusses the relationships between these systems , but for this chapter one need only understand that use-site variance is more expressive than declaration-site variance and discards the implicit constraints of wildcards, since the complications of implicit constraints far outweigh their usefulness [Tate, Leung, and Lerner, 2011].

In our simplified formalism, classes/interfaces $\mathcal{C}$ have exactly one type parameter; all the rules, algorithms, and proofs will be extended to arbitrary type parameters in Section 2.5.5. More importantly, when supplying a type argument to a class/interface, one provides both an **in** bound and an **out** bound. In terms of arrays, the **in** bound is what can be put into the array, and the **out** bound is what can be taken out of the array. Formally, the **in** bound is the argument to the contravariant portion of the class/interface and the **out** bound is the argument to the covariant portion of the class/interface. We also use $\bot$ and $\top$ as the subtype and supertype of all types. That way Java's $\mathcal{C}\langle$`? extends` $\tau\rangle$ can translate to $\mathcal{C}\langle$**in** $\bot$ **out** $\tau\rangle$, and $\mathcal{C}\langle$`? super` $\tau\rangle$ to $\mathcal{C}\langle$**in** τ **out** $\top\rangle$.

2.3.1 *Materials*

Materials are the classes/interfaces exchanged between separate components of a program and stored within the components of a program. More

formally, they are the classes/interfaces that are used as parameter and return types for functions/methods/constructors as well as types of fields. Consequently, most classes/interfaces are materials.

Supposing $\mathcal{M}$ is the subset of classes/interfaces $\mathcal{C}$ that are materials, we define the grammar of our parameterized types as follows:

$$\dot{\tau} ::= \bot \mid \top \mid \mathcal{M}\langle \textbf{in } \dot{\tau} \textbf{ out } \dot{\tau} \rangle \mid \cdot$$

The $\cdot$ represents the single parameter of the inheriting type; a complete discussion of type variables appears in Section 2.5.4.

Observe that we make parameterized types $\dot{\tau}$ be comprised of only materials. However, any class/interface can inherit any other; only the type arguments are restricted to materials. Thus we formalize inheritance as a relationship of the following form:

$$\mathcal{C}\langle \cdot \rangle <:: \mathcal{C}'\langle \dot{\tau}' \rangle$$

We do not impose a grammar for specifying inheritance, rather we leave that to the language and assume it provides one. Consequently, we demand the following three properties in order to accurately model inheritance:

TRANSITIVITY

$$\mathcal{C}\langle \cdot \rangle <:: \mathcal{C}'\langle \dot{\tau}' \rangle \quad \wedge \quad \mathcal{C}'\langle \cdot \rangle <:: \mathcal{C}''\langle \dot{\tau}'' \rangle$$
$$\Downarrow$$
$$\mathcal{C}\langle \cdot \rangle <:: \mathcal{C}''\langle \dot{\tau}''[\cdot \mapsto \dot{\tau}'] \rangle$$

FINITENESS For all classes/interfaces $\mathcal{C}$ and $\mathcal{C}'$, the set of parameterized types $\dot{\tau}'$ such that $\mathcal{C}\langle \cdot \rangle <:: \mathcal{C}'\langle \dot{\tau}' \rangle$ holds is finite.

ACYCLICITY There is no $\mathcal{C}$ and $\dot{\tau}'$ such that $\mathcal{C}\langle \cdot \rangle <:: \mathcal{C}\langle \dot{\tau}' \rangle$ holds.

Typically this relationship will be derived from some simpler one via transitive closure, but we require transitivity in order to simplify many of our formalisms. Nonetheless, with a little care one can easily reformulate our system for a non-transitive inheritance relationship. On a related note, we will use $\leq::$ to denote the reflexive closure of $<::$.

2.3.2 *Shapes*

Shapes capture the recursive aspects of inheritance and are the reason we need F-bounded polymorphism [Canning et al., 1989] rather than just plain bounded polymorphism. For example, the common Java interface `Comparable` is a shape because classes such as `Integer` implement `Comparable`$\langle$*Integer*$\rangle$, a type defined in terms of the class/interface inheriting it. In current practice, only a few classes/interfaces are shapes, but those classes/interfaces are often used widely throughout the project.

From our observations, shapes arise in practice for two main reasons. The primary one is to encode a form of self types [Bruce, Odersky, and Wadler, 1998]. That is, the type parameter of the shape is meant to represent the type implementing that shape. This is useful for binary methods, such as comparisons and equalities, as well as algebraic operations, such as addition, negation, and multiplication. Negation is an important example because it illustrates that self types are not just used for binary methods.

The second use of shapes is type families [Ernst, 2001]. A type family is a codependent group of classes/interfaces. A classic example is graphs, edges, and vertices. A graph consists of edges and vertices; edges connect vertices and reside within a graph; and vertices have connecting edges

```
interface Graph⟨G extends Graph⟨G,E,V⟩
                E extends Edge⟨G,E,V⟩,
                V extends Vertex⟨G,E,V⟩⟩ {
    List⟨V⟩ getVertices();
}
interface Edge⟨G extends Graph⟨G,E,V⟩,
               E extends Edge⟨G,E,V⟩,
               V extends Vertex⟨G,E,V⟩⟩ {
    G getGraph();
    V getSource();
    V getTarget();
}
interface Vertex⟨G extends Graph⟨G,E,V⟩,
                 E extends Edge⟨G,E,V⟩,
                 V extends Vertex⟨G,E,V⟩⟩ {
    G getGraph();
    List⟨E⟩ getIncoming();
    List⟨E⟩ getOutgoing();
}

class Map extends Graph⟨Map,Road,City⟩ {...}
class Road extends Edge⟨Map,Road,City⟩ {...}
class City extends Vertex⟨Map,Road,City⟩ {...}
```

Figure 2.1: A type family for graphs, edges, and vertices

and reside within a graph. The challenge is designing this group such that when one extends it, say with mutability, then all components can refer to the other components and know they are also mutable. To accomplish this with shapes, each interface takes three type parameters, one for graphs, one for edges, and one for vertices, and all bounded to indicate so. Extensions of the type family then impose additional constraints on the type parameters to indicate the guaranteed additional functionality. We illustrate this design pattern in Figure 2.1.

To formalize the recursive nature of inheritance with shapes, we first define a labeled graph describing how classes/interfaces are used in inheritance:

$$\frac{\mathcal{C}\langle\,\cdot\,\rangle <:: \mathcal{C}'\langle\dot{\tau}'\rangle}{\mathcal{C} \to \mathcal{C}'} \qquad \frac{\mathcal{C}\langle\,\cdot\,\rangle <:: \mathcal{C}'\langle\dot{\tau}'\rangle \qquad \mathcal{C}'' \text{ occurs in } \dot{\tau}'}{\mathcal{C} \xrightarrow{\mathcal{C}'} \mathcal{C}''}$$

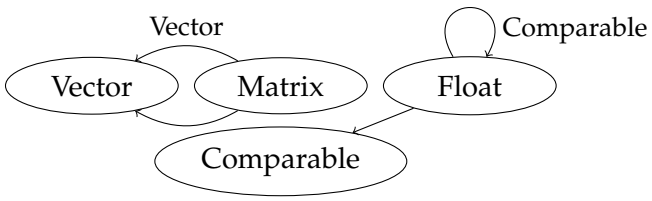
If one were to require classes/interfaces to inherit only types that are already defined, then this usage graph would be acyclic, and subtyping can be proven decidable by using a topological ordering of the classes/interfaces. However, in a system with recursive inheritance, such a topological ordering does not exist. Shapes $\mathcal{S}$ are the classes/interfaces such that if the edges labeled with shapes were removed from the usage graph then it would be acyclic. Thus shapes are the classes/interfaces preventing the topological ordering that would make subtyping easily decidable.

A common misconception, partly based on the common examples for them, is that shapes are interfaces and materials are classes. It is worth stressing that both can be either, and whether something must be a shape or not is solely determined by how it is used in the inheritance hierarchy.

As an example, consider the following class declarations.

```
interface Comparable⟨E⟩ {}
class     Vector⟨E⟩ {}
class     Matrix⟨E extends Comparable⟨E⟩⟩
          extends Vector⟨Vector⟨E⟩⟩{}
class     Float extends Comparable⟨Float⟩ {}
```

These result in the following usage graph.



The unlabeled edge from `Matrix` to `Vector` is due to the direct extension `class Matrix extends Vector⟨...⟩` and the labeled edge is due to

`Vector⟨X⟩` being the type argument to `Vector⟨...⟩` in that extension. The `Float` class has a self loop labeled `Comparable`, creating a cycle in the usage graph containing `Comparable`, indicating that `Comparable` is a shape. Note that the constraint on `Matrix`'s parameter `E` has no effect on this graph; we discuss the role of type variables in Section 2.5.4.

2.3.3 *Separating Materials and Shapes*

While it is theoretically possible to have a class/interface be used as both a material and a shape, our aforementioned interaction with developers suggests there is a natural tendency to keep these two patterns separate. Here we formalize that assumption.

Material-Shape Separation. *Let $\mathcal{M}$ be all classes/interfaces used as type arguments. For some set $\mathcal{S}$ of classes/interfaces such that removing all edges labeled with an element of $\mathcal{S}$ from the usage graph results in an acyclic graph, $\mathcal{M}$ and $\mathcal{S}$ are disjoint.*

Although this formalization does not need $\mathcal{M}$ and $\mathcal{S}$ to cover $\mathcal{C}$, it is convenient to simply define $\mathcal{M}$ as all non-shapes. In this way, unless a programmer explicitly declares a class/interface to be a shape, they are free to use that class/interface without restriction outside the class/interface hierarchy.

Under this design, our `List` example is still rejected, but for more intuitive reasons. First, the designer would specify that `Equatable` is a shape. Then, when defining `List`, our system would indicate that `Equatable` cannot be used as an argument to `List` due to being a shape. Hence the cause and effect are clear to the designer, who can then focus on finding a

type-safe alternative. Regrettably, this leaves our problem with `Equatable` unsolved, which we defer to future work as discussed in Section 2.6, but it prevents programmers from creating unconventional designs, assuming that such designs are indeed unconventional, which we verify in the next section.

2.4 INDUSTRY COMPATIBILITY

To support our claim that Material-Shape Separation captures an industry-wide idiom, we present the findings of our scientific inquiry into current practices. Over 13.5 million lines of generic-Java code across a total of 62 open-source projects, taken primarily from the Qualitas Corpus [Tempero et al., 2010], show no alarming cases where separation was broken. A table of all projects we analyzed and some relevant statistics we collected can be found in Figure A.2 in Appendix A. Projects ranged in scale and function from the jFin finance library to the massive NetBeans IDE, the median size being approximately 60,000 lines of code. As such, our sample set contains a wide range of styles and design principles. Nevertheless, the projects conformed to our system, suggesting that one can enforce Material-Shape Separation in existing languages, as well as in new ones, without breaking compatibility with existing code bases.

2.4.1 *Methodology*

After forming our collection of projects, we modified the source code of `openjdk` to generate the usage graphs of Section 2.3.2 from the classes/in-

terfaces of each project. From these graphs, we extracted the labels of the edges constituting simple cycles. These labels formed our set of shapes $\mathcal{S}$, and all other classes/interfaces formed our set of materials $\mathcal{M}$. Another compiler pass then searched for occurrences, if any, of shapes being used as materials, thereby violating Material-Shape Separation.

2.4.2 *Findings*

Barring a few caveats discussed below, the entire body of 62 projects never violated Material-Shape Separation. In fact, every shape we encountered was either an encoding of self types or type families, as we had expected. The type family we encountered happened to be precisely for representing graphs. In the `findbugs` project, interfaces `GraphVertex` and `GraphEdge`, and classes `AbstractVertex` and `AbstractEdge`, constituted type families at the interface level, and at the class level, similar to the design in Figure 2.1. Some custom shapes, `hadoop`'s `WritableComparable` as well as `findbugs`'s `AnnotationEnumeration`, are simple extensions of `Comparable`. In fact, `WritableComparable` is actually just an encoding of an intersection type, which will be discussed in Section 2.5.2. As for self types, `Comparable` and `Enum` are the two incorporated into Java's libraries and are the most widely used. Moreover, the remaining nine remaining shapes were all custom applications of self types. All counted, there were 17 shapes in total, listed in Figure A.1 in Appendix A, none of which were used as materials.

Caveats

The above statements make a few simplifications, namely eliding technicalities caused by programmer errors and Java limitations. First, there were uses of the shapes outside of inheritance and type-variable constraints. However, all these uses were in the form of raw types (except in one case where the type argument was simply an unconstrained wildcard, thereby not utilizing the type argument). That is, the programmers used shapes as materials only when bypassing Java's type system, sacrificing type safety. These were either results of poor utilization of generics (e.g. failing to use F-bounded polymorphism in order to ensure type safety) or involved casting wherein Java can only enforce raw types due to type erasure [Gosling, Joy, Steele, and Bracha, 2005].

Because none of these uses of shapes as materials actually used their type argument, it is still possible to incorporate them into our system. For each shape, we can associate a new parameterless material inherited by the shape. This material is not inherited recursively, so it is not a shape. We can substitute all the above raw (or wildcarded) misuses of the shape with the new parameterless material inherited by that shape. Thus, since the arguments of shapes are never used in the code bases, through this encoding they still all satisfy Material-Shape Separation. Regardless, it is better to view these few instances as abuses of the type system rather than as reflective of design principles.

The second caveat is due to the following class in `openjdk`:

```
public class Env⟨A⟩ implements Iterable⟨Env⟨A⟩⟩ {}
```

Because of this one class, we originally inferred `Iterable` to be a shape, even though this inheritance clause is never actually made use of by the

code base nor exposed by the API and so should not have been present. `Iterable` is used widely as a material, so this inference caused many false alarms, demonstrating the danger of inferring shapes rather than having them be explicitly identified by the programmer.

2.4.3 *Ceylon*

One might be surprised by how few shapes we discovered in use: roughly one shape per million lines of code. However, every shape had a key and distinct role in its respective architecture design. We simply have recognized these as special cases and classified their distinction. Nonetheless, one might worry that our observations may not persist over more designs given the limited sample we draw our conclusions from here. Similarly, our observations might only apply to Java because of the burden Java imposes upon using generics. To address this issue, we have adapted our analysis to Ceylon, a language recently designed and released by Red Hat that fully embraces generics. Self types and type families are directly supported by Ceylon, and Ceylon uses shapes to support features such as operator polymorphism [King, 2013]. Thus, shapes appear much more frequently in Ceylon than in Java, providing a denser sample.

We presented Material-Shape Separation and our corresponding results to the Ceylon team. They found the analysis and applications compelling and simple enough that within a day they had implemented a branch of their compiler that enforced Material-Shape Separation. They decided to treat precisely the self types and type families as shapes instead of using our inference technique. They used the modified compiler on all

the committed code that had been developed in the language, either by the designers implementing core modules or by contributors adding new modules to the open-source project, and found only one counterexample to Material-Shape Separation. This counterexample was a labeled-tree design similar to the problematic `Tree` example to be discussed in Section 2.5.1. It was a quickly-drafted practical implementation of a JSON API, and its design was already in contention at the time. Furthermore, this instance is easily resolved by adding a `children` attribute to the class in place of the extension clause, similar to the example we include in Section 2.6. The designers have continued to confirm that unconstrained programmers still naturally adhere to Material-Shape Separation even with their more expressive type system. Their current stance is that they will likely integrate Material-Shape Separation into Ceylon 2.0.

2.5 APPLICATIONS

Having introduced the formal definitions of materials and shapes and demonstrated their compatibility with existing code bases, we now describe how we can exploit our newfound Material-Shape Separation to design simple, sound, and complete type-checking algorithms. This section presents five results immediately realizable through shapes: the decidability of subtyping, the support for non-syntactic type equivalence, the existence of joins, the ability to constrain type variables, and the incorporation of higher-kinded types. In Section 2.6, we will discuss additional existing challenges and new features we hope to address in future work by extending the techniques we present here.

2.5.1 *Decidability of Subtyping*

Recall the example `List` design:

```
interface List⟨out E⟩
        extends Equatable⟨List⟨Equatable⟨E⟩⟩⟩ {}
```

`javac`[2] handles most uses of this design correctly. However, this design violates both Kennedy and Pierce's and Tate et al.'s restrictions on generics [Kennedy and Pierce, 2007; Tate, Leung, and Lerner, 2011], and consequently we can use it to cause `javac` to stack overflow.

Consider the following use of the `List` design:

```
class Tree extends ArrayList⟨Tree⟩ {}
```

In one line, it implements a mutable unlabeled tree. Furthermore, since `ArrayList` implements `List`, we also get the correct equality implementation for trees with no additional effort. But upon actually equating two trees, `javac` throws a `StackOverflowError`.

Understanding why the type checker fails is crucial to understanding the surprising challenges behind generics. To check a use of the equality operation, the type checker needs to verify that the left type implements `Equatable` of the right type. Here this reduces to checking that `Tree` is a subtype of `Equatable⟨Tree⟩`. This simple question evolves into the infinite progression of subtyping reductions shown in Figure 2.2.

Note that the final state of the above is the same as the initial state, forming a loop that causes the infinite digression. What is surprising is that this infinite digression corresponds to a *valid* infinite proof of subtyping (refer to Tate et al. for more details [Tate, Leung, and Lerner, 2011]). These

2 When we refer to `javac` we mean the OpenJDK `1.7.0_25` type checker.

$$
\begin{aligned}
\texttt{Tree} &<: \texttt{Equatable}\langle\texttt{Tree}\rangle \\
&\Downarrow \ (\textit{inheritance}) \\
\texttt{ArrayList}\langle\texttt{Tree}\rangle &<: \texttt{Equatable}\langle\texttt{Tree}\rangle \\
&\Downarrow \ (\textit{inheritance}) \\
\texttt{List}\langle\texttt{Tree}\rangle &<: \texttt{Equatable}\langle\texttt{Tree}\rangle \\
&\Downarrow \ (\textit{inheritance}) \\
\texttt{Equatable}\langle\texttt{List}\langle\texttt{Equatable}\langle\texttt{Tree}\rangle\rangle\rangle &<: \texttt{Equatable}\langle\texttt{Tree}\rangle \\
&\Downarrow \ (\textit{contravariance}) \\
\texttt{Tree} &<: \texttt{List}\langle\texttt{Equatable}\langle\texttt{Tree}\rangle\rangle \\
&\Downarrow \ (\textit{inheritance}) \\
\texttt{ArrayList}\langle\texttt{Tree}\rangle &<: \texttt{List}\langle\texttt{Equatable}\langle\texttt{Tree}\rangle\rangle \\
&\Downarrow \ (\textit{inheritance}) \\
\texttt{List}\langle\texttt{Tree}\rangle &<: \texttt{List}\langle\texttt{Equatable}\langle\texttt{Tree}\rangle\rangle \\
&\Downarrow \ (\textit{covariance}) \\
\texttt{Tree} &<: \texttt{Equatable}\langle\texttt{Tree}\rangle \\
&\vdots
\end{aligned}
$$

Figure 2.2: Infinite Progression of Subtyping Reductions

infinite proofs are what make subtyping so difficult to decide, since they imply that an algorithm can in fact make good progress in each step but still never be able to finish. However, with Material-Shape Separation, all proofs of subtyping are finite, so any such algorithm is guaranteed to terminate.

To demonstrate this, we first formalize *extended* types σ. Extended types are not used in practice, but we can guarantee decidable subtyping even for extended types, so we present them here to provide more informed options for language designers.

$$\sigma := \bot \mid \top \mid \mathcal{C}\langle \textbf{in}\ \sigma\ \textbf{out}\ \sigma\rangle$$

$\dot{\tau}$	$\dot{\tau}[\cdot \mapsto \sigma_i; \sigma_o]$
$\bot$	$\bot$
$\top$	$\top$
$\cdot$	σ_o
$\mathcal{M}\langle \textbf{in } \dot{\tau}_i \textbf{ out } \dot{\tau}_o \rangle$	$\mathcal{M}\langle \textbf{in } \dot{\tau}_i[\cdot \mapsto \sigma_o; \sigma_i] \textbf{ out } \dot{\tau}_o[\cdot \mapsto \sigma_i; \sigma_o] \rangle$

Subtyping $\vdash \sigma <: \sigma$

$$\vdash \bot <: \sigma \qquad \vdash \sigma <: \top$$

$$\frac{\mathcal{C}\langle \cdot \rangle \leq:: \mathcal{C}'\langle \dot{\tau}' \rangle \qquad \vdash \sigma_i' <: \dot{\tau}'[\cdot \mapsto \sigma_o; \sigma_i] \qquad \vdash \dot{\tau}'[\cdot \mapsto \sigma_i; \sigma_o] <: \sigma_o'}{\vdash \mathcal{C}\langle \textbf{in } \sigma_i \textbf{ out } \sigma_o \rangle <: \mathcal{C}'\langle \textbf{in } \sigma_i' \textbf{ out } \sigma_o' \rangle}$$

Figure 2.3: Algorithmic subtyping rules

The primary difference between σ and $\dot{\tau}$ is that σ allows arbitrary classes/interfaces $\mathcal{C}$ rather than just materials $\mathcal{M}$. Consequently, extended types may use shapes, even as type arguments. The intuition behind this is that, for subtyping, our separation of materials and shapes need only be imposed upon the class/interface hierarchy and not on types elsewhere in the program. The second difference is that σ is not parameterized; we will address the issue of type variables shortly in Section 2.5.4.

Figure Figure 2.3 formalizes subtyping on extended types. The rule for subtyping classes/interfaces combines inheritance and use-site variance into one step. The subtlety here is substitution, described in the table in Figure Figure 2.3 above the subtyping rules, which has to deal with the fact that there are two type arguments for a single type parameter. This substitution replaces all contravariant uses of the type parameter with the **in** argument, and all covariant uses with the **out** argument. This technique combines the subtyping and tight-approximation algorithms of Tate [2013] into one rule.

The subtyping rules are syntax directed and so specify a sound and complete decision algorithm *provided* we can guarantee the process terminates. Our finiteness assumption on $<::$ prevents infinite branching at any point. Consequently, the only remaining source of non-termination is the potential for infinite proofs, much like in our example earlier. This brings us to our main theorem.

Theorem 2.1. *Under Material-Shape Separation, all proofs of subtyping as specified in Figure 2.3 are finite.*

Proof. The major insight is that Material-Shape Separation implies that new uses of shapes are never introduced when applying inheritance in subtyping since shapes can never occur in the type arguments of inherited classes/interfaces. Thus we can define a well-founded two-part measure on extended types σ.

The first part, $\lfloor \sigma \rfloor$ formalized in Figure 2.4, is the maximum layering depth of shapes in the extended type, where a layer is a shape occurring syntactically inside a type argument to another shape. This part of the measure is completely agnostic to the inheritance hierarchy, since we know that inheritance cannot introduce new layers of shapes so long as it satisfies Material-Shape Separation. Thus recursion in inheritance causes no problems.

The second part, $|\sigma|$ formalized in Figure 2.4, specifies the maximum number of proof steps that can be taken from any situation where σ is on either side of a subtyping judgement until reaching a shape at the top level (thereby next reducing the first part of the measure) or terminating. The challenge is to prove that this measure is well defined; in other words, the calculation $|\sigma|$ must terminate. This is clear by structural induction

σ	$\lfloor\sigma\rfloor : \mathbb{N}$
$\bot$	0
$\top$	0
$\mathcal{M}\langle\textbf{in } \sigma_i \textbf{ out } \sigma_o\rangle$	$\max(\lfloor\sigma_i\rfloor, \lfloor\sigma_o\rfloor)$
$\mathcal{S}\langle\textbf{in } \sigma_i \textbf{ out } \sigma_o\rangle$	$1 + \max(\lfloor\sigma_i\rfloor, \lfloor\sigma_o\rfloor)$

$\sigma/\dot{\tau}$	$\lvert\sigma/\dot{\tau}\rvert : \mathbb{N}/\dot{\mathbb{N}}$
$\cdot$	$\cdot$
$\bot$	0
$\top$	0
$\mathcal{M}\langle\textbf{in } \sigma_i/\dot{\tau}_i \textbf{ out } \sigma_o/\dot{\tau}_o\rangle$	$1 + M_{\mathcal{M}}[\,\cdot \mapsto \max(\lvert\sigma_i/\dot{\tau}_i\rvert, \lvert\sigma_o/\dot{\tau}_o\rvert)]$
$\mathcal{S}\langle\textbf{in } \sigma_i/\dot{\tau}_i \textbf{ out } \sigma_o/\dot{\tau}_o\rangle$	0

$$M_{\mathcal{M}} = \max(\,\cdot\,, \max_{\mathcal{M}\langle\,\cdot\,\rangle <:: \mathcal{M}'\langle\dot{\tau}\rangle} M_{\mathcal{M}'}[\,\cdot \mapsto |\dot{\tau}|])$$

We implicitly lift max *and* $1+$ *to parameterized integers.*

Figure 2.4: Measures for extended/parameterized types

provided each parameterized measure $M_{\mathcal{M}}$ is well defined. The parameterized measure $M_{\mathcal{M}}$ is a function on measures indicating how applying inheritance affects the measure of a $\mathcal{M}$ type. The key observation is that $M_{\mathcal{M}}$ only uses the parameterized measures of inherited *materials*. Due to Material-Shape Separation, well foundedness of material inheritance enables us to assume that those parameterized measures are already well defined, thereby making $M_{\mathcal{M}}$ well defined.

This two-part measure on types can be adapted into a measure on subtyping judgements. We define the measure of a judgement $\vdash \sigma <: \sigma'$ as the lexicographic ordering $(\lfloor\sigma\rfloor + \lfloor\sigma'\rfloor)$ followed by $(|\sigma| + |\sigma'|)$. One can easily verify that for each rule the measure of the premises is always strictly less than the measure of the conclusion, thereby guaranteeing that any proof will be finite even if infinite proofs were permitted. □

Corollary 2.1. *Under Material-Shape Separation, subtyping as specified in Figure 2.3 is decidable.*

What is remarkable about this result is that our subtyping algorithm is more naïve than prior solutions and yet is still both sound and complete under Material-Shape Separation. For example, Kennedy and Pierce's prohibition against expansive inheritance does not prevent infinite proofs; it only ensures all infinite proofs eventually cycle thanks to results from Viroli [2000]. Therefore, their algorithm requires keeping a list of all the subtyping judgements that arose earlier in the recursion stack and checking them against the current judgement for syntactic identity before proceeding to process the judgement as usual in order to determine if they are in an infinite cyclic proof [Kennedy and Pierce, 2007]. While not as computationally difficult, Tate, Leung, and Lerner [2011] prevent infinite recursion by treating invariant types as a special case using syntactic unification. Notice that both these approaches rely on syntactic identity, whereas we only use recursion, which brings us to our next contribution.

2.5.2 *Equivalences*

Syntactic identity of types can be troublesome for type systems in which there are multiple ways to express the same type. In practice, this has not been a large problem because many existing type systems have the property that all equivalent types are syntactically identical. However, newer and more expressive languages cannot rely on syntactic identity. Tate, Leung, and Lerner [2011] presented some issues with this flawed assumption in Java, illustrating that semantically equivalent types can be

written differently and that consequently `javac` rejects programs due to such shallow syntactic differences. Tate, Leung, and Lerner [2011]'s own algorithm actually also relies on syntactic identity, so they describe a complex multipass process for canonicalizing types. Ideally such complications would not be necessary because they can be rather brittle and sensitive to changes in the language design. Our system has no such problem. Syntactic identity is never used in our subtyping algorithm, so type equivalences are already incorporated and decidable.

To describe the circumstances more formally, let us suppose we make the following extension to our types:

$$\begin{aligned} \dot{\tau} &::= \cdots \mid \mathcal{M}\langle !\dot{\tau}\rangle \\ \sigma &::= \cdots \mid \mathcal{C}\langle !\sigma\rangle \end{aligned}$$

The ! annotation indicates an invariant usage of the type argument. In many systems, this is the default. We previously used only the **in** and **out** arguments because ! represents the special case where both arguments are the same; however, existing type systems are more accurately formalized with an ! annotation.

In such systems, subtyping is specified with the following additional rules:

$$\frac{\mathcal{C}\langle\,\cdot\,\rangle \leq:: \mathcal{C}'\langle\dot{\tau}'\rangle \qquad \vdash \sigma'_i <: \dot{\tau}'[\,\cdot \mapsto \sigma] \qquad \vdash \dot{\tau}'[\,\cdot \mapsto \sigma] <: \sigma'_o}{\vdash \mathcal{C}\langle !\sigma\rangle <: \mathcal{C}'\langle \textbf{in}\ \sigma'_i\ \textbf{out}\ \sigma'_o\rangle}$$

$$\frac{\mathcal{C}\langle\,\cdot\,\rangle \leq:: \mathcal{C}'\langle\dot{\tau}'\rangle \qquad \dot{\tau}'[\,\cdot \mapsto \sigma] = \sigma'}{\vdash \mathcal{C}\langle !\sigma\rangle <: \mathcal{C}'\langle !\sigma'\rangle}$$

The second rule uses syntactic identity. This is the status quo in many type systems and algorithms, but it does not interact well with other type features.

To demonstrate the problem, suppose we were to add intersection types. To do so, we would make the following extensions to our system:

$$\dot{\tau} \quad ::= \cdots \mid \dot{\tau} \cap \dot{\tau}$$

$$\sigma \quad ::= \cdots \mid \sigma \cap \sigma$$

$$\frac{\vdash \sigma_i <: \sigma'}{\vdash \sigma_1 \cap \sigma_2 <: \sigma'} \qquad \frac{\vdash \sigma <: \sigma'_1 \qquad \vdash \sigma <: \sigma'_2}{\vdash \sigma <: \sigma'_1 \cap \sigma'_2}$$

With intersection types one can require a field to be both iterable and serializable using the type `Iterable⟨T⟩&Serializable`. Such a type is not expressible in Java, and consequently programmers often opt to leave the `Serializable` requirement implicit and manually cast when necessary, somewhat defeating the purpose of a static type system.

Given such a feature, one might eventually obtain an object of type `Array⟨Iterable⟨T⟩&Serializable⟩`, where `Array` is an invariant type (we have slipped to declaration-site variance for sake of clarity). Similarly, a function might need an object of type `Array⟨Serializable&Iterable⟨T⟩⟩`. The question is whether the former can be used for the latter. The answer seems to be obviously yes, since $\cap$ is a commutative operator, but the type systems and algorithms using syntactic identity would reject such a coercion since the two intersections are written differently. At first this might seem easy to fix, but the problem is subtler than it appears. In particular, `Serializable` and `Iterable` have no direct connection to each

other, making this is an easy example. But we could also have the type `Iterable⟨T⟩&List⟨T⟩⟨,⟩` in which case the left is a supertype of the right and therefore redundant. That is, `Iterable⟨T⟩&List⟨T⟩⟨`i`⟩`s equivalent to `List⟨T⟩`. Thus determining equivalences of intersections relies on subtyping, and determining proper subtyping relies on determining equivalences, producing a circularity.

This troublesome circularity is best illustrated with the following class definition:

```
class Foo extends Array⟨Foo&Array⟨Foo⟩⟩ {}
```

Now consider whether `Foo` is a subtype of `Array⟨Foo⟩`. The subtyping holds iff `Foo&Array⟨Foo⟩⟨i⟩`s equivalent to `Foo`, and that equivalence holds iff `Foo` is a subtype of `Array⟨Foo⟩`. Thus we have a circular dependency, so we can answer yes to both or no to both and in either case we have a consistent system. This situation is due to the problematic infinite proofs we discussed earlier.

Fortunately, having observed Material-Shape Separation, we recognize that the above example is impractical and need not be addressed. Moreover, our encoding of invariant types replaces the rule using syntactic identity with the following rule:

$$\frac{\mathcal{C}\langle\cdot\rangle \leq:: \mathcal{C}'\langle\dot{\tau}'\rangle \qquad \vdash \sigma' <: \dot{\tau}'[\cdot \mapsto \sigma] \qquad \vdash \dot{\tau}'[\cdot \mapsto \sigma] <: \sigma}{\vdash \mathcal{C}\langle !\sigma\rangle <: \mathcal{C}'\langle !\sigma'\rangle}$$

Hence our system already uses type equivalence rather than syntactic identity. Plus, our strategy for guaranteeing all proofs are finite easily extends to incorporate intersections. Put together, these properties give a

sound and complete subtyping algorithm with intersections that uses type equivalence rather than syntactic identity.

2.5.3 *Joins*

Whereas our subtyping results applied to extended types, our remaining findings only apply to non-extended types τ:

$$\tau ::= \bot \mid \top \mid \mathcal{M}\langle \textbf{in}\ \tau\ \textbf{out}\ \tau \rangle$$

Given a pair of types τ_1 and τ_2, their join $\tau_1 \sqcup \tau_2$ is their most-precise common supertype. Joins are useful for the type checker, particularly in operations that combine expressions. For example, consider the following program:

```
⟨T extends Comparable⟨in T⟩⟩
void separate(T middle,
              Iterable⟨out T⟩ elems,
              ArrayList⟨in T⟩ smaller,
              ArrayList⟨in T⟩ bigger) {
   foreach (T elem in elems)
      (elem < middle ? smaller : bigger).add(elem);
}
```

Each element of the list is added to `smaller` or `bigger` depending on how it compares with `middle`. To type check `? :`, though, one needs to combine the types of `smaller` and `bigger` into a common supertype. If the most precise such common supertype is computed, then the subsequent method call `.add(elem)` is rejected only if the program is invalid.

In this case, such a most-precise common supertype seems easy to determine since the types being joined are in fact the same. However,

we chose this example because it both arose from practice and broke `javac`. The program, once translated into Java's syntax, is valid but `javac` incorrectly rejects it.

The reason is that `javac` uses an imprecise join algorithm that discards any uses of **`? super`** (i.e. **`in`**) in the types being joined. It does so because Java's type system does not have joins, and even when they exist they can be difficult to determine. For this reason, Smith and Cartright proposed simply adding *union* types [Smith and Cartwright, 2008], trivially guaranteeing joins because the rules for union types actually define them as the join of the types being unioned together. However, such a fix is shallow, since then one needs to extend all other type-checking rules to handle union types. For example, Tate et al. demonstrate that Smith and Cartwright's approach does not address capture conversion [Tate, Leung, and Lerner, 2011], an important feature for using wildcards with generic methods [Torgersen et al., 2004]. Tate et al. instead use lazy existential types as a regrettably complex solution.

As an example of the intricacies of this problems, suppose we need to join together the two simple types `Integer` and `Float`. To simplify matters, further suppose that `Integer` only implements `Summable⟨Integer⟩` and `Float` only implements `Summable⟨Float⟩`. One common supertype of `Integer` and `Float` is `Summable⟨?⟩`, but so is `Summable⟨`**`out`**` Summable⟨?⟩⟩` and `Summable⟨`**`out`**` Summable⟨`**`out`**` Summable⟨?⟩⟩⟩`, and each is more precise than the one before it. In fact, we can continue this chain forever, demonstrating that there is no most-precise common supertype for this simple practical example. That is, the join of `Integer` and `Float` does not exist.

Now we apply Material-Shape Separation to this problem. Notice that `Summable` is a shape; it appears in two cases of recursive inheritance.

Consequently, `Summable`$\langle ? \rangle$ is not a valid type τ in our system because `Summable` is not a material. In fact, none of the above common supertypes are valid types in our system. `Summable` is only permitted in inheritance and type-variable constraints, not as the type of an expression. Thus in our system the join of `Integer` and `Float` is simply $\top$. The following proof demonstrates that all joins are similarly easy to compute in our system, provided we have intersection types.

Theorem 2.2. *Under Material-Shape Separation, our type system extended with intersection types has computable joins for all types τ_1 and τ_2 with respect to other types τ.*

Proof. The algorithm is the following: (1) if either τ_j is $\bot$, then the join is τ_k where $j \neq k$; (2) if either τ_j is $\top$, then the join is $\top$; (3) if either τ_j is $\tau \cap \tau'$, then the join is $(\tau \sqcup \tau_k) \cap (\tau' \sqcup \tau_k)$ where $j \neq k$; (4) otherwise, each τ_j must be of the form $\mathcal{M}_j\langle \mathbf{in}\ \tau_i^j\ \mathbf{out}\ \tau_o^j \rangle$, and the join is

$$\bigcap_{\substack{\mathcal{M}_1\langle\cdot\rangle \leq :: \mathcal{M}'\langle \dot{\tau}_1' \rangle, \\ \mathcal{M}_2\langle\cdot\rangle \leq :: \mathcal{M}'\langle \dot{\tau}_2' \rangle}} \mathcal{M}' \left\langle \begin{array}{ll} \mathbf{in} & \dot{\tau}_1'[\,\cdot \mapsto \tau_o^1; \tau_i^1] \cap \dot{\tau}_2'[\,\cdot \mapsto \tau_o^2; \tau_i^2] \\ \mathbf{out} & \dot{\tau}_1'[\,\cdot \mapsto \tau_i^1; \tau_o^1] \sqcup \dot{\tau}_2'[\,\cdot \mapsto \tau_i^2; \tau_o^2] \end{array} \right\rangle$$

This algorithm can easily be shown to terminate reusing the second component of the measure used for subtyping. Once again, well-foundedness of material inheritance is the critical feature. Note that the large intersection only ranges over inherited materials rather than all classes/interfaces, which is safe to do because other types τ are only comprised of materials. This is how we avoid the issue of recursive inheritance via shapes.

The key step for proving the algorithm correct is proving that joins distribute through intersections. Ignoring $\bot$ and $\top$ at the moment for simplicity, any type τ is essentially of the form $\bigcap_i \tau_i$ where each τ_i is an instantiation of some material and i ranges over some finite number. Given two such types $\bigcap_i \tau_i$ and $\bigcap_k \tau''_k$, it is easy to prove that $\bigcap_i \tau_i <: \bigcap_k \tau''_k$ can only hold if for all k there exists some i such that τ_i is a subtype of τ''_k. So, if $\bigcap_k \tau''_k$ is a common supertype of $\bigcap_i \tau_i$ and $\bigcap_j \tau'_j$, then for each k there exists some i and some j such that τ''_k is a common supertype of τ_i and τ'_j. Thus $\bigcap_k \tau''_k$ is a common supertype of $\bigcap_{i,j} \tau_i \sqcup \tau'_j$, from which the result follows. □

The reader might take issue with our use of intersection types, which allowed us to avoid computing the *meet*, or least-precise common subtype of two types. Indeed, many languages impose restrictions on multiple inheritance, and unrestricted intersection types can be used to violate invariants that would otherwise hold such as single-instantiation inheritance for arbitrary types. Additional subtleties surrounding intersection types include uninhabitable intersections, which a precise type system would replace with $\bot$. These issues are rather specific to details of a given language design, so their discussion lies outside the scope of this chapter, but we have found Material-Shape Separation to be useful in these settings. Here we used unrestricted intersection types because they are necessary for handling arbitrary multiple inheritance.

Note that, although in the case of subtyping we only provided an alternative to existing approaches to guaranteeing decidability, in the case of joins none of those existing approaches guarantee the existence, let

alone the computability, of joins. Even the simple example before with `Integer` and `Float` proved problematic in those systems.

2.5.4 *Type Variables and Constraints*

So far we have managed to avoid the issue of type variables, a rather important concept given the topic of F-bounded polymorphism. We did so because we can view type variables simply as abstract classes/interfaces. Upper-bound constraints on type variables translate to inheritance clauses on these type variables. There is a technical issue with the use of top-level use-site variance permitted in constraints but not in inheritance clauses, but this is purely grammatical and easy to accommodate. Lower-bound constraints on type variables can sometimes translate to locally adding inheritance clauses to the constraining class/interface.

To illustrate our perspective, recall the code from Section 2.3.2:

```
interface Comparable⟨E⟩ {}
class     Vector⟨E⟩ {}
class     Matrix⟨E extends Comparable⟨E⟩⟩
          extends Vector⟨Vector⟨E⟩⟩ {}
class     Float extends Comparable⟨Float⟩ {}
```

Inside the body of `Matrix`, the type variable `E` is in scope. Various types will reference `E`, and subtyping will need to take its constraint into account. To integrate this into our formalism, note that if `E` were a class/interface C with the inheritance clause `E` <:: `Comparable⟨E⟩` then Material-Shape Separation would still hold. Thus, subtyping will still be decidable. If `E` had a lower bound such as `Integer`, then subtyping would still be decidable since

$$\kappa ::= * \mid \langle \overline{\kappa} \rangle \to \kappa$$
$$\tau ::= \bot \mid \top \mid X \mid \mathcal{M} \mid \tau\langle \overline{\textbf{in } \tau \textbf{ out } \tau} \rangle \mid \lambda \overline{X}.\tau$$
$$\Theta ::= \overline{X : \kappa}$$

Type Validity $\Theta \vdash \tau : \kappa$

$$\Theta \vdash \bot : * \qquad \Theta \vdash \top : * \qquad \frac{X : \kappa \in \Theta}{\Theta \vdash X : \kappa}$$

$$\frac{\mathcal{M} : \langle \overline{\kappa} \rangle \to *}{\Theta \vdash \mathcal{M} : \langle \overline{\kappa} \rangle \to *} \qquad \frac{\Theta, \overline{X : \kappa} \vdash \tau : \kappa'}{\Theta \vdash \lambda \overline{X}.\tau : \langle \overline{\kappa} \rangle \to \kappa'}$$

$$\frac{\Theta \vdash \tau : \langle \overline{\kappa} \rangle \to \kappa' \qquad \overline{\Theta \vdash \tau_i : \kappa} \qquad \overline{\Theta \vdash \tau_o : \kappa}}{\Theta \vdash \tau\langle \overline{\textbf{in } \tau_i \textbf{ out } \tau_o} \rangle : \kappa'}$$

Figure 2.5: Higher-kinded types

adding the inheritance clauses `Integer` <:: `E` and (transitively required) `Integer` <:: `Comparable`⟨`E`⟩ still satisfies Material-Shape Separation.

Note that a lower bound such as `Vector`⟨`Integer`⟩ cannot be translated like above into our formalization of inheritance, so our current proof does not extend to such lower bounds. However, our formalism could be extended to handle such lower bounds by treating them like inheritance clauses when generating the usage graph (extending the definition of Material-Shape Separation) and when defining the measure of variables (extending our proof strategy). The one caveat is that shapes cannot be used in lower bounds for this strategy to work. Indeed, if `Foo` inherited `Shape`⟨`Foo`⟩ and `X` had lower bound `Shape`⟨**out** `X`⟩, then there would be an infinite proof that `Foo` is a subtype of `X`.

2.5.5 *Higher Kinds*

With this strategy of viewing type variables as abstract classes/interfaces, we can extend type variables to having higher kinds since class/interface names are essentially higher-kinded types. One could declare a parameterized type variable `C⟨X⟩` and require it to extend `Iterable⟨X⟩` so that `C` represents some iterable generic class/interface. One could even further constrain `CX⟨t⟩`o extend `Equatable⟨C⟨X⟩⟩` so as to ensure this kind of collection comes with a semantics and decision algorithm for equality. One only needs to prove that higher-kinded subtyping [Pierce and Steffen, 1997] is decidable.

Unfortunately, higher-kinded subtyping is not decidable with extended types. To understand why, consider the following definitions (using declaration-site variance for the sake of convenience):

```
shape    Shape⟨in P : ∗⟩ {}
material Mayhem⟨Q : ∗ → ∗⟩
         extends Shape⟨Q⟨Mayhem⟨Q⟩⟩⟩ {}
```

Note that `Shape` itself has kind $* \to *$, so `Mayhem⟨Shape⟩` is a valid type of kind $*$. Consider, then, whether `Mayhem⟨Shape⟩` is a subtype

Type Application $\tau \rightsquigarrow \tau$

$$(\lambda \overline{X}.\tau)\langle \overline{\mathbf{in}\ \tau_i\ \mathbf{out}\ \tau_o} \rangle \rightsquigarrow \tau[\overline{X \mapsto \tau_i; \tau_o}]$$

$$\frac{\tau \rightsquigarrow \tau'}{\tau\langle \overline{\mathbf{in}\ \tau_i\ \mathbf{out}\ \tau_o} \rangle \rightsquigarrow \tau'\langle \overline{\mathbf{in}\ \tau_i\ \mathbf{out}\ \tau_o} \rangle}$$

Higher-Kinded Subtyping $\Theta \vdash \tau <: \tau : \kappa$ / $\Theta \vdash \tau <: \mathcal{S}\langle \overline{\mathbf{in}\ \tau\ \mathbf{out}\ \tau} \rangle : *$

$$\Theta \vdash \bot <: \tau : * \qquad \Theta \vdash \tau <: \top : *$$

$$\frac{X : \overline{\langle \overline{\kappa} \rangle \to} * \in \Theta \qquad \overline{\overline{\Theta \vdash \tau_i' <: \tau_i : \kappa}} \qquad \overline{\overline{\Theta \vdash \tau_o <: \tau_o' : \kappa}}}{\Theta \vdash X\,\overline{\langle \overline{\mathbf{in}\ \tau_i\ \mathbf{out}\ \tau_o} \rangle} <: X\,\overline{\langle \overline{\mathbf{in}\ \tau_i'\ \mathbf{out}\ \tau_o'} \rangle} : *}$$

$$\frac{\mathcal{M} : \langle \overline{\kappa} \rangle \to * \qquad \mathcal{C} : \langle \overline{\kappa} \rangle \to * \qquad \mathcal{M}\langle \overline{X} \rangle \leq:: \mathcal{C}\langle \overline{\tau} \rangle \qquad \overline{\overline{\Theta \vdash \tau_i' <: \tau[\overline{X \mapsto \tau_o; \tau_i}] : \kappa}} \qquad \overline{\overline{\Theta \vdash \tau[\overline{X \mapsto \tau_i; \tau_o}] <: \tau_o' : \kappa}}}{\Theta \vdash \mathcal{M}\langle \overline{\mathbf{in}\ \tau_i\ \mathbf{out}\ \tau_o} \rangle <: \mathcal{C}\langle \overline{\mathbf{in}\ \tau_i'\ \mathbf{out}\ \tau_o'} \rangle : *}$$

$$\frac{\tau \rightsquigarrow \hat{\tau} \qquad \Theta \vdash \hat{\tau} <: \tau' : *}{\Theta \vdash \tau <: \tau' : *} \qquad \frac{\tau' \rightsquigarrow \hat{\tau}' \qquad \Theta \vdash \tau <: \hat{\tau}' : *}{\Theta \vdash \tau <: \tau' : *}$$

$$\frac{\Theta, \overline{X : \kappa} \vdash \tau\langle \overline{\mathbf{in}\ X\ \mathbf{out}\ X} \rangle <: \tau'\langle \overline{\mathbf{in}\ X\ \mathbf{out}\ X} \rangle : \kappa'}{\Theta \vdash \tau <: \tau' : \langle \overline{\kappa} \rangle \to \kappa'}$$

Figure 2.6: Subtyping rules for higher-kinded types

of $\texttt{Shape}\langle\texttt{Mayhem}\langle\texttt{Shape}\rangle\rangle$. We can prove this with the following infinite derivation (making intermediate steps explicit):

$$\begin{aligned}
\texttt{Mayhem}\langle\texttt{Shape}\rangle &<: \texttt{Shape}\langle\texttt{Mayhem}\langle\texttt{Shape}\rangle\rangle \\
&\Downarrow \ (\textit{inheritance}) \\
\texttt{Shape}\langle\texttt{Shape}\langle\texttt{Mayhem}\langle\texttt{Shape}\rangle\rangle\rangle &<: \texttt{Shape}\langle\texttt{Mayhem}\langle\texttt{Shape}\rangle\rangle \\
&\Downarrow \ (\textit{contravariance}) \\
\texttt{Mayhem}\langle\texttt{Shape}\rangle &<: \texttt{Shape}\langle\texttt{Mayhem}\langle\texttt{Shape}\rangle\rangle \\
&\vdots
\end{aligned}$$

This example exploits the fact that `Shape` can be used as an argument to a higher-kinded parameter that can be used without restriction in order to violate our invariant that shapes are never introduced by expanding inheritance.

Fortunately, due to Material-Shape Separation we can adapt our earlier proof strategy to a higher-kinded type system without nested shapes. We formalize this higher-kinded type system in Figure 2.5 and its subtyping rules in Figure 2.6. For the sake of algorithmic simplicity, we present the minimal form of type-level computation necessary for the system to work as expected; this minimality is not necessary for our proof strategy below. We use $\bar{}$ to indicate "some number of", being consistent with that unknown number across multiple uses of $\bar{}$ within a rule, and similarly for $\bar{\bar{}}$. For example, in the rule for variables, $\bar{\bar{}}$ represents the number of applications to the variable X, and $\bar{}$ represents the number of arguments in each of those applications. The premises indicate that each corresponding pair of **in** (or **out**) arguments of each corresponding pair of applications must be supertypes (or subtypes).

Theorem 2.3. *Under Material-Shape Separation, all proofs of subtyping as specified in Figure 2.6 are finite.*

Proof. As before, our strategy is to identify a measure for types such that the sum of the measures of the types being compared always decreases as the syntax-directed algorithm progresses. We no longer need to consider nested uses of shapes, but now we must consider higher-kinded types. To do so, we assign a type τ of kind κ a measure $|\tau|$ of type $\kappa[* \mapsto \mathbb{N}]$. For example, a type $\hat{\tau}$ of kind $\langle *, * \rangle \to *$ is assigned a measure $|\hat{\tau}|$ of type $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$. The intuition is that if $\hat{\tau}$ were applied to types with measures m and n then $|\hat{\tau}|\,(m, n)$ is the measure of the applied type.

We define this measure in Figure 2.7 (reusing type-variable names as measure-variable names). The challenge is to prove that this measure is well defined. To do so, observe that the definition of the measure function $M_{\mathcal{M}}$ for a material constructor $\mathcal{M}$ only references measure functions for materials used in the inheritance clauses of $\mathcal{M}$. Due to Material-Shape Separation, we can assume that those material functions are terminating, thereby making $M_{\mathcal{M}}$ a terminating function as well. Structural induction then easily demonstrates that the measure is well defined on all types.

This measure on types can be adapted into a measure on subtyping judgements. We define the measure of a subtyping judgement $\overline{X : \kappa} \vdash \tau <: \tau' : *$ to be $(|\tau| + |\tau'|)[\overline{X \mapsto 0}]$. Note that we only define this measure for subtyping judgements of kind $*$. This is because we view the only rule applicable to other kinds as intermediate since by structural induction on the kind it simply introduces fresh variables until the types being compared have kind $*$.

Finally, one can easily verify that for each rule the measure of the premises (after processing the rule for introducing fresh variables) is al-

$\tau : \kappa$	$\lvert\tau\rvert : \kappa[* \mapsto \mathbb{N}]$
$\bot$	0
$\top$	0
X	X
$\mathcal{M}$	$M_{\mathcal{M}}$
$\tau\langle\overline{\textbf{in } \tau_i \textbf{ out } \tau_o}\rangle$	$1 + \lvert\tau\rvert\,(\overline{\max(\lvert\tau_i\rvert, \lvert\tau_o\rvert)})$
$\lambda\overline{X}.\tau$	$\lambda\overline{X}.\lvert\tau\rvert$

$$M_{\mathcal{M}:\langle\overline{\kappa}\rangle\to *} = \lambda\overline{X}.\max\left(\max(\overline{\zeta_\kappa(X)}), \max_{\mathcal{M}\langle\overline{X}\rangle <:: \mathcal{M}'\langle\overline{\tau}\rangle} M_{\mathcal{M}'}(\overline{|\tau|})\right)$$

$$\text{where} \qquad \zeta_*(m) = m \qquad \zeta_{\langle\overline{\kappa}\rangle\to\kappa'}(m) = \zeta_{\kappa'}(m)(\overline{0})$$

We implicitly lift max, $1+$, *and* 0 *when applied to functions.*

Figure 2.7: Measure for higher-kinded types

ways strictly less than the measure of the conclusion, thereby guaranteeing that any proof must be finite even if infinite proofs were permitted. □

The proof for computable joins extends similarly. Thus, by separating materials and shapes we are able to add a powerful, fully functional feature to our type system with minimal effort.

2.6 FUTURE WORK

In this chapter, we have shown that the separation of materials and shapes is practical, with a broad survey demonstrating its compatibility with existing code and with anecdotes offering insight into why this pattern arises. We have also shown that this separation simplifies and improves various core typing algorithms even in the presence of intersection types and higher-kinded type variables. Now we present new type features that may be made possible by the results of this chapter.

2.6.1 *Conditional Inheritance*

Although Material-Shape Separation solves a number of open type-checking problems, our initial motivating use case remains unsolved. Recall that we wanted a type-safe way to make `Lists` have equality whenever their elements have equality. We believe we could apply our findings to conditional inheritance to produce an effective solution. Here is how our example might look like using conditional inheritance:

```
interface  List⟨out T⟩ {...}
           extends Equatable⟨List⟨T⟩⟩
           given T extends Equatable⟨T⟩ {...}
```

This seems ideal, something akin to Haskell's type classes [Wadler and Blott, 1989], but now consider our `Tree` specification once again. The question again is whether `Tree` extends `Equatable⟨Tree⟩`. Since `Tree` extends `List⟨Tree⟩`, this holds provided `List⟨Tree⟩` extends `Equatable⟨Tree⟩`. By the above specification, `List⟨Tree⟩` extends `Equatable⟨List⟨Tree⟩⟩` (a subtype of `Equatable⟨Tree⟩`) *provided* `Tree` extends `Equatable⟨Tree⟩`. And now we are back where we started. Once again we are building a valid, yet infinite proof.

cJ and JavaGI have already made an effort to incorporate conditional inheritance [Huang, Zook, and Smaragdakis, 2007; Wehr, Lämmel, and Thiemann, 2007]. However, cJ has no proof for decidable type checking (and appears to be undecidable), and the above example causes the JavaGI compiler to stack overflow even though the language has a proof of decidability. Most likely this is because a decision algorithm for the above would at least need to track all entailments and subtypings currently being

processed and continually check these for repeats in order to identify when inside a cyclic infinite proof, a rather expensive and complicated process. Even if implemented correctly, this approach is most likely brittle and may not be able to extend to systems where type equivalence does not imply syntactic identity.

We believe conditional inheritance would be decidable in our system. In particular, we disallow the problematic recursive specification of `Tree` in Section 2.5.1, instead encouraging the following:

```
class  Tree extends Equatable⟨Tree⟩ {
       List⟨Tree⟩ children() {...}
       Boolean equals(Tree that) {
           return children().equals(that.children());
       }
}
```

This implementation provides predictable, understandable behavior. Furthermore, in the case of shapes, we believe it would be possible to override a default implementation locally without ever producing any semantic inconsistency via variance and subtyping, since shapes may not occur as type arguments. Nonetheless, there are many subtleties to explore both in terms of type checking and in terms of run-time implementation, so we defer detailed investigation to future work.

2.6.2 *Decidable Intraprocedural Type Inference*

With computable joins, we have the beginnings of decidable intraprocedural type inference. Ideally one would be able to take a function whose context is well established, including types for parameters and an explicit return type, and determine whether it type checks without needing any

type annotations in its body. There are two major challenges we foresee for completing this goal. First, one must design an object-oriented type system with principal types, which requires addressing practical issues such as overloading, as well as theoretical issues such as type-argument inference (see Chapter 8). Second, one must infer the types of loop variables whenever typeable. This latter challenge may prove very difficult since, even given Material-Shape Separation, subtyping is still not well founded despite all proofs being finite. For example, `Array`⟨**in** `Object`⟩ is a subtype of `Array`⟨**in** `Array`⟨**in** `Array`⟨**in** `Object`⟩⟩⟩, which is only the beginning of an infinite progression. Nonetheless, Material-Shape Separation drastically simplifies the forms that subtyping constraints can take, so we believe it may be a first step towards decidable intraprocedural type inference for object-oriented languages. Such a feature would not only make programming in statically-typed languages more convenient, but also enable easier optimizations in gradual typing: it may enable us to try to type-check untyped functions when they are cast at run time to see if we can optimize them based on their inferred types.

2.6.3 *Virtual Types*

Virtual types are, in summary, the idea that objects can have types as members [Kristensen et al., 1983; Madsen and Møller-Pedersen, 1989]. For example, each graph object could have a member `V` indicating the type of its own vertices. Self types are a special form of virtual types, and type families are a means to approximate virtual types with F-bounded polymorphism.

There are many ways to implement the concept of virtual types within a type system [Bruce, Odersky, and Wadler, 1998; Igarashi and Pierce, 1999; Odersky and Zenger, 2005; Thorup, 1997; Thorup and Torgersen, 1999; Torgersen, 1998][3]. For example, with significant effort one can encode virtual types using the implicit constraints [Tate, Leung, and Lerner, 2011] of Java's wildcards combined with wildcard capture [Torgersen et al., 2004], or much more simply one can use Scala's path-dependent types [Odersky and Zenger, 2005]. Regardless of the specifics, any encoding of virtual types must address their many subtleties, such as those relating to wildcards as described by Tate et al. [Tate, Leung, and Lerner, 2011]. We posit that Material-Shape Separation may alleviate these subtleties. For example, by incorporating the constraints on the virtual types of a class/interface into the usage graph and measure, we might be able to extend the definition of Material-Shape Separation and the proof of decidable subtyping to virtual types. With further investigation, one might be able to support constrained virtual types without sacrificing principles such as decidability.

2.7 RELATED WORK

Previous work in this area has focused primarily on algorithmic issues. Kennedy and Pierce mapped the boundary of decidable subtyping, giving three forms of restrictions each of which would guarantee decidability [Kennedy and Pierce, 2007]. These provided subsequent works, ours included, a firm basis for future explorations.

3 Since the underlying paper was published, Zhang, Loring, et al. [2015] separated shapes and materials more clearly by distinguishing contraints and class definitions syntactically. Their follow-on work on family polymorphism [Zhang and Myers, 2017] implements a powerful version of virtual types, and posits decidability based on Material-Shape Separation, though without proof.

Wehr et al. built JavaGI, adding conditional inheritance to the type system [Wehr, Lämmel, and Thiemann, 2007], and incorporating Kennedy and Pierece's results to achieve the decidability missing from cJ [Huang, Zook, and Smaragdakis, 2007]. However, probably due to the complexity of the underlying algorithms, the implementation of their type checker does not match the specification. Our results suggest that acknowledging the separation between materials and shapes might help to repair and simplify their implementation.

Smith and Cartwright identified problems specific to type-argument inference in Java and proposed an extension to the type system with corresponding algorithms [Smith and Cartwright, 2008]. In particular, Java wildcards do not admit joins, so Smith and Cartwright proposed adding union types to Java's type system, though these introduce complications elsewhere in the type system [Tate, Leung, and Lerner, 2011]. Our finding is that, by not allowing shapes as type arguments, we admit and can compute joins without the need for union types. This change could be incorporated into Smith and Cartwright's algorithms.

Most recently, Tate et al. identified nested contravariance as a source of complications [Tate, Leung, and Lerner, 2011]. Removing it, they found, would make subtyping decidable in a manner compatible with existing code bases. Yet their restrictions significantly restrict contravariance and are strongly influenced by corner cases. Like Smith and Cartwright, Tate et al.'s proposal does not admit joins, nor does it extend to the many features we have addressed in this chapter.

2.8 SUMMARY

This chapter explained a key insight about how programmers use F-bounded polymorphism, and how to use this insight to prove that the straightforward subtyping algorithm for Java terminates (so long as Material-Shape separation is observed). This result is a key foundation to the rest of the work discussed in this dissertation; as we discussed earlier, it is important for gradual typing that run-time type checks – which in our case are subtyping checks – terminate and are reasonably efficient. The computability of joins is also important, as meets and joins become relevant in type-argument inference, something that will come up again in Chapter 8.

On top of that, the system is simple to understand, implement, and extend. The barrier for adopting the separation of materials and shapes is very low, especially when contrasted with the gains in both decidability and simplicity, we believe that it can easily be incorporated into new and existing statically-typed object-oriented languages. As evidence, the designers of Ceylon have already taken interest in this design, and are likely to integrate Material-Shape Separation into Ceylon 2.0.

In the next chapter, we show how a decidable algorithm like the one presented here can be extended to extend to union and intersection types and many kinds of interactions between the two and other type-system features.

3

INTEGRATED SUBTYPING

This chapter is based on a paper presented at OOPSLA 2018: *Empowering Union and Intersection Types with Integrated Subtyping* [Muehlboeck and Tate, 2018b]. It comes with a Coq formalization that can be found in the supplementary materials of this dissertation and on the ACM website [Muehlboeck and Tate, 2018a]. This formalization is an implementation of the framework discussed here and can be used as a library to plug in your own type system definitions just as discussed in the rest of this chapter.

3.1 INTRODUCTION

In many languages, types can be interpreted as sets of values. In a language with subtyping, subtypes can often be interpreted as subsets, and union and intersection types—which can be interpreted as unions and intersections of sets—are an easy way to obtain joins and meets in the subtyping hierarchy. Thus union and intersection types have often been used in tasks like type inference [Gosling, Joy, Steele, and Bracha, 2005] and static analysis [Palsberg and Pavlopoulou, 1998; Wells et al., 2002]. However, until recently they have rarely been exposed directly to programmers.

This chapter is phrased completely independently of gradual typing, as its results are a lot more widely and immediately applicable elsewhere. As mentioned before, the main motivation from a gradual typing point of view is that decidable subtyping for intersections and unions means trivially obtainable meets and joins, which may become useful for generic type-argument inference, see also Part III.

Part of the problem is that the set-based model suggests certain interactions of union and intersection types with each other and the rest of the type system, but these interactions have established themselves to be

difficult to implement in ways that are both decidable and extensible. For example, one would expect unions and intersections to be distributive, but the standard syntax-directed subtyping rules for union and intersection types cannot recognize the following subtyping:

$$\tau \cap (\tau_1 \cup \tau_2) <: (\tau \cap \tau_1) \cup (\tau \cap \tau_2)$$

Pierce [1991] explored combining union and intersection types in the context of the Forsythe programming language [Reynolds, 1988, 1997]. He gave rules for distributivity of intersections over both unions and functions, but he also relied on an explicit transitivity rule and thus had no clear decidable algorithm for subtyping, leaving this goal for future work. Decidability in this particular setting has been established in work on *minimal relevant logic* [Gochet, Gribomont, and Rossetto, 2005; Routley and Meyer, 1972; Viganò, 2000] and on semantic subtyping [Frisch, Castagna, and Véronique Benzaken, 2002]. However, the algorithms and proofs of these works are not easy to generalize and adapt into even slight extensions of the type systems. As an example, for nearly a decade the Scala team has investigated adding true union types to the language but has yet to make such an addition. The Dotty prototype in development for Scala does support union types [Rompf and Amin, 2016; The Dotty Development Team, 2015], but not in decidable manner, and it will probably be some more years until it matures to the point of supporting union types with "enough" decidability for practical purposes. Flow [Facebook, 2014] and Pony [Clebsch et al., 2015] do have union and intersection types, but with very limited reasoning that is unable to recognize aspects such as distributivity. Typed Racket [Tobin-Hochstadt and Felleisen, 2008], on

the other hand, can recognize distributivity, but cannot recognize deeper properties such as the fact that a pair of union types is equivalent to a union of pair types. Julia only has union types but is capable of such deeper reasoning [Bezanson et al., 2017; Zappa Nardelli et al., 2018]. However, it is unknown whether Julia subtyping is decidable or even transitive, which are particularly important questions for Julia since subtyping is heavily used in its operational semantics, and which we discuss in more detail near the end of this chapter. TypeScript [Bierman, Abadi, and Torgersen, 2014; Microsoft, 2012] is best situated to take advantage of prior research due to its heavy use of structural subtyping, which has been the focus of semantic subtyping [Ancona and Corradi, 2016; Castagna and Xu, 2011; Frisch, Castagna, and Véronique Benzaken, 2002, 2008; Hosoya and Pierce, 2003]. However, in order to accept common patterns in the JavaScript community, TypeScript chose to make its subtyping system intentionally unsound and intentionally intransitive [Microsoft, 2018, Type Compatibility]. Lack of soundness means TypeScript can be optimistically aggressive with how it reasons about union and intersection types. Lack of transitivity means this aggressive reasoning can be inconsistent and hard to predict. For example, TypeScript will recognize that a particular value f belongs to a particular function type $(x : \tau_i) \Rightarrow \tau_o$ but then reject an invocation of f with an argument of type τ_i even when the required reasoning can be soundly conducted using the conservative techniques we describe here. Thus, even with the recent rise of union and intersection types in industry, there is still significant room for improvement, especially along the front of principled subtyping algorithms.

In this chapter, we show how to extend reasoning about intersection types in a decidable and extensible manner. The resulting technique has

already been adopted by Ceylon [King, 2013], which, based on this work, was able to implement several type-system features on top of union and intersection types (Section 3.2). For example, Ceylon's type-checker can recognize that an intersection like `String` ∩ `Int` is equivalent to the bottom type `Nothing`, as `String` and `Int` have no common instances, and even uses this to implement pattern matching.

The key idea presented in this chapter is that modifying the subtyping algorithm for naïve union and intersection types to *integrate* types as it recurses can significantly improve its ability to recognize desirable subtyping relationships (Section 3.4). Furthermore, provided this integration operation on types satisfies various requirements, this improved subtyping algorithm is mechanically verified in Coq [The Coq Development Team, 1984] to be sound and complete with respect to an extended subtyping system for union and intersection types, one in which union and intersection types are empowered with useful interactions with other aspects of the type system. And by composing integration operators, we can repeatedly extend the system with more and more reasoning capabilities (Section 3.5). With this toolkit, we were able to apply our framework to the Ceylon programming language, which opted to fully integrate union and intersection types into its design due to the expressiveness and guarantees we were able to provide them with (Section 3.6).

While we motivate the framework with challenges posed by the Ceylon team, it is language independent. In fact, as we detail the context of our framework (Section 3.3), we will use Forsythe's function types as our running example. Furthermore, as union and intersection types provide a simple way to obtain joins and meets in a type system, they can be useful in generic type-argument inference, as discussed in Chapter 8,

provided, of course, that they integrate well with subtyping, which we establish here. We close the chapter with a broader discussion of the generality of the framework and its connections to other research and languages (Section 3.7).

3.2 MOTIVATION

Before discussing how our framework works, we first illustrate what our framework makes possible. We do so by demonstrating the impact it had on the Ceylon programming language. All of the features discussed in this section were features the Ceylon design team wanted to provide, but were only willing to do so if the features could be implemented in a simple and reliable manner. In each case, we realized the feature could naturally be encoded with union and intersections types *provided* union and intersection subtyping could be made sufficiently intelligent, and by encoding them all into subtyping we could make the features interact in a principled manner. Unfortunately the required intelligence was significantly beyond what the state of the art in union and intersection subtyping could achieve. Thus we created our framework to provide a unified algorithm for all of these features, enabling predictable, well-defined, and powerful interactions between them, by empowering union and intersection subtyping for Ceylon.

PRINCIPAL TYPES The Ceylon team wanted every expression to have a type that best describes it, a concept known as principal types (which should not be confused with principal type schemes [Damas and Milner,

1982]) or as minimal types [Curien and Ghelli, 1992]. This property makes it much easier to provide tooling for the language, such as efficient IDE support. However, principal types are not easy to ensure in a language with both subtype polymorphism and parametric polymorphism.

To see why, consider the following generic method:

```
Iterable⟨E⟩ concat⟨E⟩(Iterable⟨E⟩ first, Iterable⟨E⟩ second)
```

It is safe to call this with an `Iterable⟨Int⟩` and an `Iterable⟨Float⟩` as the first and second arguments. The reason is that `Iterable` is *covariant*, so both types are subtypes of `Iterable⟨Number⟩`. The problem is that deriving this common supertype, `Number`, is not always easy, especially since one needs the *least* common-supertype in order to provide the principal type. For example, in Java the least common-supertype of `Int` and `Float` is actually an infinite type because both classes are `Comparable` with themselves. And to make matters worse, similar examples with *contravariant* generic classes and interfaces require computing the *greatest* common-*sub*type of two types.

Union and intersection types seem to make this problem trivial: the union type `Int` ∪ `Float` is *by definition* the least common-supertype of `Int` and `Float`, and the intersection of two types is by definition their greatest common-subtype. But while union and intersection types do easily solve the original problem, they introduce whole new problems.

To see why, suppose we have two type variables `A` and `B`. Consider what the resulting type of passing an `Iterable⟨A⟩` ∩ `Iterable⟨B⟩` to the following generic method should be:

```
E first⟨E⟩(Iterable⟨E⟩ elems)
```

One valid type is `A`, since $\texttt{Iterable}\langle\texttt{A}\rangle \cap \texttt{Iterable}\langle\texttt{B}\rangle$ is by definition at least an $\texttt{Iterable}\langle\texttt{A}\rangle$. But for that same reason, another valid type is `B`. Unfortunately, neither valid type is more precise than the other, so they both fail to be principal types.

The only possible principal type for this example is $\texttt{A} \cap \texttt{B}$. But one must be careful. In C#, this would be unsound because C# permits *multiple*-instantiation inheritance. This means a class `Foo` can implement $\texttt{Iterable}\langle\texttt{Int}\rangle$ using one set of methods and $\texttt{Iterable}\langle\texttt{Float}\rangle$ using another set. Consequently, a call to `first` with a `Foo` argument will necessarily pick one of the two implementations of `Iterable` and return a value that is either an `Int` *or* a `Float` but not both.

On the other hand, Java and Ceylon both disallow multiple-instantiation inheritance. And due to their restrictions on inheritance, they each can safely admit the following subtyping rules:

$$C\langle\tau\rangle \cap C\langle\tau'\rangle <: C\langle\tau \cap \tau'\rangle \quad \text{when } C \text{ is covariant}$$
$$C\langle\tau\rangle \cap C\langle\tau'\rangle <: C\langle\tau \cup \tau'\rangle \quad \text{when } C \text{ is contravariant}$$

By using the first of these rules, one can show that $\texttt{A} \cap \texttt{B}$ is in fact the principal type of the expression in question. In general, these extensions to subtyping can be used to easily deduce principal types for many uses of generic methods (though not all, since such principal types do not always exist).

DISJOINTNESS AND NULL-SAFETY Another feature Ceylon wanted to provide was the ability to recognize when intersections are uninhab-

itable. For example, `String` and `Int` have no common instances, so the intersection `String` $\cap$ `Int` is uninhabitable. More precisely, it is semantically equivalent to the bottom type `Nothing`. In Java, this type, if it existed, would be inhabited by **`null`**. Ceylon is null-safe, meaning types explicitly indicate whether or not they can be inhabited by **`null`**, so `Nothing` is in fact uninhabitable.

The utility of this feature is actually best demonstrated by its interaction with null-safety. In Ceylon, one uses the type `String?` to represent values that are either strings or **`null`**. This `?` operator is just a shorthand, and in fact `String?` simply represents the union type `String` $\cup$ `Null`.

Now suppose you have a list of `String?` and you want to fetch the non-null elements of the list. Ideally you could do so polymorphically using a generic method whose type argument can always be inferred, and in Ceylon this is achieved by the following generic method:

```
Iterable⟨E ∩ Object⟩ filterNulls⟨E⟩(Iterable⟨E⟩ elems)
```

Note that the return type is `Iterable`$\langle$`E` $\cap$ `Object`$\rangle$. If we provide this method with a list of `String?`, the instantiation of the return type in full will be `Iterable`$\langle$`(String` $\cup$ `Null)` $\cap$ `Object`$\rangle$. Since Ceylon is null-safe, every value of `Object` is necessarily not null, and every `String` is already an `Object`, so the type system should ideally be able to recognize that this type is equivalently just an `Iterable`$\langle$`String`$\rangle$. It can do so by recognizing distributivity of intersections over unions as well as the following subtyping rule that is sound in any language with single-class inheritance:

$C \cap C' <: \bot$ when C and C' are classes and neither inherits the other

EXHAUSTIVE PATTERN MATCHING The Ceylon team wanted to support algebraic data types with exhaustive pattern matching. They also wanted to support being able to branch based on the run-time type of a value. And ideally these related concepts could be handled uniformly.

The following is an example of how such a uniform treatment can be utilized:

```
switch (nums) { // nums has type Int∪LinkedList⟨Int⟩
  case (is Int) { return nums; }
  case (is Nil) { return 0; }
  case (is Cons) { return nums.head; }
} // expected return type is Int
```

Here `nums` is either an integer or a linked list of integers. The `LinkedList` class is declared with a clause "**of** `Nil, Cons`" that restricts it to have only two subclasses: `Nil` and `Cons`. Consequently, the type system should ideally be able to deduce that this pattern match is in fact exhaustive.

Rather than require a separate set of rules for exhaustiveness, our framework was able to make subtyping powerful enough to answer this question with a simple subtyping check. In particular, we just have to check if the type of `nums` is a subtype of the union of all the cases. In this example, we can prove this subtyping so long as we can recognize the following subtyping rule:

$$C\langle\tau\rangle <: C_1\langle\tau\rangle \cup \cdots \cup C_n\langle\tau\rangle \quad \text{when } C \textbf{ of } C_1, \ldots, C_n$$

FLOW-SENSITIVITY Lastly, the Ceylon design team wanted the language to be flow-sensitive. This would avoid the tedious and unreliable pattern of **instanceof** checks followed by casts. Naïvely, one could achieve this by simply changing the type of the variable to the type in the

`instanceof` (or **`is`**) check. However, this loses all other information about the variable that might have been in the context. So a better technique is to intersect the type of the variable with the type in the check.

The last case of the above pattern match illustrates how this typing rule, in combination with an improved subtyping system, successfully unifies the features we have discussed. In this case, the type of `nums` is refined to be

$$(\texttt{Int} \cup \texttt{LinkedList}\langle\texttt{Int}\rangle) \cap \texttt{Cons}\langle *\rangle$$

where $*$ indicates the type argument is unknown since it was not checked in the cast. By distributivity, this is a subtype of

$$(\texttt{Int} \cap \texttt{Cons}\langle *\rangle) \cup (\texttt{LinkedList}\langle\texttt{Int}\rangle \cap \texttt{Cons}\langle *\rangle)$$

By disjointness, the left intersection is a subtype of $\bot$, which effectively eliminates this case of the union, leaving us with

$$\texttt{LinkedList}\langle\texttt{Int}\rangle \cap \texttt{Cons}\langle *\rangle$$

Since `LinkedList` has an **`of`** clause, the left type is a subtype of the union of its cases, resulting in

$$(\texttt{Nil}\langle\texttt{Int}\rangle \cup \texttt{Cons}\langle\texttt{Int}\rangle) \cap \texttt{Cons}\langle *\rangle$$

And by distributivity and disjointness again, we can eliminate the `Nil` case, leaving us with only the $\texttt{Cons}\langle\texttt{Int}\rangle$ case. Consequently, we can determine that `nums` is specifically a $\texttt{Cons}\langle\texttt{Int}\rangle$, informing us that it indeed has a `head` field with the expected return type `Int`.

The framework we developed is able to address each of these problems in a principled manner. Furthermore, the framework prescribes how to compose each of the individual solutions together into one coherent subtyping algorithm simultaneously supporting all the extensions, as needed by the above example. Next we describe when in general our framework can be applied, and how it achieves these results. As our running example, we will use the much simpler concept of function types and our application to Forsythe. But afterwards, we will revisit the above more elaborate collection of features desired by Ceylon.

3.3 FORMALIZING TRADITIONAL UNION AND INTERSECTION SUBTYPING

Our goal is to empower subtyping in type systems with union and intersection types. Thus we are looking at type systems where types have the following general form:

$$\tau ::= \bot \mid \top \mid \tau \cup \tau \mid \tau \cap \tau \mid \ell$$

The special types $\bot$ and $\top$ form the bottom and top of the subtyping hierarchy, respectively, while unions $\cup$ and intersections $\cap$ form joins and meets, respectively. Last, literals ℓ are the rest of the types in the particular type system at hand. For example, in an overly simplified functional language, literals would be defined mutually recursively with types τ as $\ell_{\text{Fun}} ::= \tau \to \tau$.

Subtyping systems for union and intersection types follow certain patterns, both in how they are declaratively specified and in how they are

Declarative Subtyping $\tau <: \tau$

$$\frac{}{\tau <: \tau} \qquad \frac{\tau_1 <: \tau_2 \qquad \tau_2 <: \tau_3}{\tau_1 <: \tau_3}$$

$$\frac{}{\tau <: \top} \qquad \frac{}{\bot <: \tau} \qquad \frac{r \in \mathcal{D} \qquad \forall \langle \tau^{\leftarrow}, \tau^{\rightarrow} \rangle \in \mathcal{P}_r.\ \ \tau^{\leftarrow} <: \tau^{\rightarrow}}{\ell_r^{\leftarrow} <: \ell_r^{\rightarrow}}$$

$$\frac{}{\tau_i <: \tau_1 \cup \tau_2} \qquad \frac{}{\tau_1 \cap \tau_2 <: \tau_i} \qquad \frac{\tau_1 <: \tau \qquad \tau_2 <: \tau}{\tau_1 \cup \tau_2 <: \tau} \qquad \frac{\tau <: \tau_1 \qquad \tau <: \tau_2}{\tau <: \tau_1 \cap \tau_2}$$

Figure 3.1: Declarative Subtyping

algorithmically decided. Even the proofs that the algorithm is sound and complete with respect to the specification exhibit common reductive patterns. In order to improve these preexisting subtyping systems, we need to understand these patterns. In the following, we illustrate these patterns, and at the same time we formalize them so that we may formally describe the requirements and guarantees of our framework.

3.3.1 *Declarative Subtyping*

Traditionally, subtyping with union and intersection types is specified as in Figure 3.1. In the start of the first row, subtyping is specified to be reflexive and transitive. In the second row are the declarative subtyping rules for union and intersection types modeling their set-theoretic intuitions. Lastly, at the end of the first row is the subtyping rule for literals. It is formalized abstractly by assuming a (usually infinite) set of declarative literal subtyping rules $\mathcal{D}$. For each rule $r \in \mathcal{D}$, there is a left literal $\ell_r^{\leftarrow}$ (the subtype), a right literal $\ell_r^{\rightarrow}$ (the supertype), and a set of premises $\mathcal{P}_r \subseteq \tau \times \tau$.

Each premise has two types that must be subtypes for the rule to be applicable. The literal subtyping rule in Figure 3.1 indicates that two literals are subtypes if there is a rule in $\mathcal{D}$ that concludes with those two literals and for which subtyping can be shown to hold for every premise.

As an example, consider our toy functional language whose literals ℓ_{Fun} are function types. Subtyping on functions is contravariant with respect to the parameter type and covariant with respect to the return type. Since there are an infinite number of potential parameter and return types, there are an infinite number of rules formulating variance of function types. However, this can be expressed concisely by just two rule *schemata* using *metavariables*, with rules resulting from instantiating the metavariables. It is common to conflate rules and rule schemata, and we ourselves do so in this chapter for the sake of simplicity, letting the reader apply context and intuition to disambiguate where necessary rather than constantly overwhelm the reader with disambiguations. As such the two rules (technically rule schemata) for variance of function types are the following:

$$\frac{\tau_i' <: \tau_i}{\tau_i \to \tau_o <: \tau_i' \to \tau_o} \qquad \frac{\tau_o <: \tau_o'}{\tau_i \to \tau_o <: \tau_i \to \tau_o'}$$

We can encode these rules in our framework by defining $\mathcal{D}_{\text{Fun}}$ to be comprised of two cases parametrized by the possible instantiations of the relevant metavariables:

$$\mathcal{D}_{\text{Fun}} ::= \mathsf{Contra}(\tau_i, \tau_i', \tau_o) \mid \mathsf{Co}(\tau_i, \tau_o, \tau_o')$$

The components of these rules are then defined as follows:

$r \in \mathcal{D}_{\text{Fun}}$	$\ell_r^{\leftarrow}$	$\ell_r^{\rightarrow}$	$\mathcal{P}_r$
$\text{Contra}(\tau_i, \tau_i', \tau_o)$	$\tau_i \to \tau_o$	$\tau_i' \to \tau_o$	$\{\langle \tau_i', \tau_i \rangle\}$
$\text{Co}(\tau_i, \tau_o, \tau_o')$	$\tau_i \to \tau_o$	$\tau_i \to \tau_o'$	$\{\langle \tau_o, \tau_o' \rangle\}$

Thus, $\mathcal{D}_{\text{Fun}}$ is the infinite set of all possible instantiations of the two rule schemata given above, and the metafunctions $\ell^{\rightarrow}$, $\ell^{\leftarrow}$, and $\mathcal{P}$ access the relevant parts of these instantiations.

Encodings aside, the advantage of the declarative subtyping rules is that one can quickly assess that they correctly capture the intended features. For one, subtyping is transitive by definition. But this same transitivity rule is often also the greatest problem with the declarative subtyping rules. The reason is that it is not *syntax-directed*. In trying to determine whether one type is a subtype of another, there are an infinite number of possible middle types that could be used to exploit transitivity. This means one cannot effectively search through the space of possible declarative subtyping proofs as a subtyping algorithm. Next we discuss the standard solution to this problem: design a new set of subtyping rules that are syntax-directed, and then prove this syntax-directed subtyping system to be equivalent to the declarative subtyping system.

3.3.2 *Reductive Subtyping*

There are many ways to develop subtyping algorithms and prove them correct. For reasons that should become clear once we discuss the proof of correctness, we use the term *reductive* to refer to the particularly common—

Reductive Subtyping $\tau \leq \tau$

$$\frac{r \in \mathcal{R} \qquad \forall \langle \tau^{\leftarrow}, \tau^{\rightarrow} \rangle \in \mathcal{P}_r.\ \ \tau^{\leftarrow} \leq \tau^{\rightarrow}}{\ell_r^{\leftarrow} \leq \ell_r^{\rightarrow}} \qquad \frac{}{\tau \leq \top} \qquad \frac{}{\bot \leq \tau}$$

$$\frac{\tau \leq \tau_i}{\tau \leq \tau_1 \cup \tau_2} \qquad \frac{\tau_i \leq \tau}{\tau_1 \cap \tau_2 \leq \tau} \qquad \frac{\tau_1 \leq \tau \qquad \tau_2 \leq \tau}{\tau_1 \cup \tau_2 \leq \tau} \qquad \frac{\tau \leq \tau_1 \qquad \tau \leq \tau_2}{\tau \leq \tau_1 \cap \tau_2}$$

Figure 3.2: Reductive Subtyping

even textbook [Pierce, 2002]—technique we formalize here. The syntax-directed reductive subtyping rules for union and intersection types are shown in Figure 3.2. The rules for reflexivity and transitivity have been removed, and a number of the rules specific to union and intersection types have been adjusted to conceptually directly incorporate the removed rules. And as before, the rest of subtyping consists of rules concerning specifically literals. However, in order to incorporate the removed rules into the literal rules, the abstract literal rule ranges over a different set $\mathcal{R}$ of *reductive* rules for literals.

For example, for our toy functional language, there is only one reductive subtyping rule on functions, combining the two declarative subtyping rules into one:

RULE

$$\frac{\tau_i' \leq \tau_i \tau_o \leq \tau_o'}{\tau_i \to \tau_o \leq \tau_i' \to \tau_o'}$$

ENCODING

$r \in \mathcal{R}_{\mathsf{Fun}}$	$\ell_r^{\leftarrow}$	$\ell_r^{\rightarrow}$	$\mathcal{P}_r$
$\mathsf{Fun}(\tau_i, \tau_i', \tau_o, \tau_o')$	$\tau_i \to \tau_o$	$\tau_i' \to \tau_o'$	$\{\langle \tau_i', \tau_i \rangle, \langle \tau_o, \tau_o' \rangle\}$

3.3.3 *Proof Search as an Algorithm*

The intent is to derive an algorithm from this new set of rules. Given a potential subtyping, proof search considers each rule that could conclude with that subtyping and recursively checks whether the premises of the rule hold. However, for this process to terminate, the reductive rules must satisfy two important properties.

3.3.3.1 *Syntax-Directedness*

Proof search must consider every rule that can apply to the potential subtyping at hand. For this to be possible, there must be a finite number of such rules. Furthermore, the process must consider every premise of the rules, so there must be a finite number of premises for each rule. These properties together are what make a system syntax-directed. Clearly they hold for the standard reductive subtyping rules, i.e. the non-literal rules for unions and intersections. Thus one only needs to show they also hold for the custom reductive literal rules.

REQUIREMENT 3.1 (SYNTAX-DIRECTEDNESS) *For every literal pair $\ell^{\leftarrow}$ and $\ell^{\rightarrow}$, the set of applicable reductive literal subtyping rules $\{r \in \mathcal{R} \mid \ell_r^{\leftarrow} = \ell^{\leftarrow} \wedge \ell_r^{\rightarrow} = \ell^{\rightarrow}\}$ must be computable and finite. For every reductive literal subtyping rule r in $\mathcal{R}$, the set of premises $\mathcal{P}_r$ must be computable and finite.*

3.3.3.2 *Well-Foundedness*

Proof search is recursive, and so one must ensure that this recursion terminates. For the standard reductive rules, one can easily see that the combined syntactic height of the two types being investigated always

decreases. Consequently, every path of recursion is guaranteed to always eventually reach a point in which the two types being compared are both literals. So once again termination comes down to a property of the custom reductive literal rules.

REQUIREMENT 3.2 (WELL-FOUNDEDNESS) *There exists a function m from pairs of types to some set M with a well-founded relation $\prec$ satisfying the following inequalities:*

$$\forall \tau_1^{\leftarrow}, \tau_1^{\rightarrow}, \tau_2^{\leftarrow}, \tau_2^{\rightarrow}. \quad \tau_1^{\leftarrow} \ll \tau_2^{\leftarrow} \ \wedge \ \tau_1^{\rightarrow} \ll \tau_2^{\rightarrow} \implies m(\tau_1^{\leftarrow}, \tau_1^{\rightarrow}) \preceq m(\tau_2^{\leftarrow}, \tau_2^{\rightarrow})$$

$$\forall \tau^{\leftarrow}, \tau^{\rightarrow}, r \in \mathcal{R}. \qquad \langle \tau^{\leftarrow}, \tau^{\rightarrow} \rangle \in \mathcal{P}_r \implies m(\tau^{\leftarrow}, \tau^{\rightarrow}) \prec m(\ell_r^{\leftarrow}, \ell_r^{\rightarrow})$$

where $\ll$ is the weakest reflexive relation satisfying

$$\forall i, \tau_1, \tau_2. \ \tau_i \ll \tau_1 \cap \tau_2 \ \wedge \ \tau_i \ll \tau_1 \cup \tau_2$$

Typically the ordering $\preceq$ has joins and a bottom element, which are then used to define m on union and intersection types. So the only aspect of the measure function m that is actually interesting is its definition on pairs of literals. This is the case for our toy functional language, where the measure space is $\mathbb{N}$ with $<$. In this case, m is defined on literals as follows:

$$m(\tau_i^{\leftarrow} \to \tau_o^{\leftarrow}, \tau_i^{\rightarrow} \to \tau_o^{\rightarrow}) = 1 + \max(m(\tau_i^{\rightarrow}, \tau_i^{\leftarrow}), m(\tau_o^{\leftarrow}, \tau_o^{\rightarrow}))$$

3.3.4 *Equivalence of Declarative and Reductive Subtyping*

Syntax-directedness and well-foundedness of the reductive rules ensure that proof search prescribes a proper algorithm. However, that algorithm

is only guaranteed to be sound and complete with respect to *reductive* subtyping, whereas the goal is to have a sound and complete algorithm for *declarative* subtyping. The standard is to bridge this gap by proving that the two subtyping systems are in fact equivalent, making a decision procedure for one also a decision procedure for the other. This proof is typically comprised of three main components.

3.3.4.1 *Reflexivity*

The first and easiest step is to prove that reductive subtyping is reflexive, since it has no rule declaring it as such. Interestingly, although well-foundedness was originally intended to guarantee termination of proof search, one can repurpose it to simplify our proof of reflexivity. In short, if for every τ one can apply reductive rules to the goal $\tau \leq \tau$ in order to reduce the problem to other goals of the form $\tau' \leq \tau'$, well-foundedness informs us that recursively applying this goal-reduction procedure is guaranteed to terminate.

As an example, suppose τ is of the form $\tau_1 \cap \tau_2$. Then the following shows how one can reduce the goal of proving $\tau_1 \cap \tau_2 \leq \tau_1 \cap \tau_2$ to the goals $\tau_1 \leq \tau_1$ and $\tau_2 \leq \tau_2$:

$$\dfrac{\dfrac{\tau_1 \leq \tau_1}{\tau_1 \cap \tau_2 \leq \tau_1} \qquad \dfrac{\tau_2 \leq \tau_2}{\tau_1 \cap \tau_2 \leq \tau_2}}{\tau_1 \cap \tau_2 \leq \tau_1 \cap \tau_2}$$

Similar reductions can be done for $\bot$, $\top$, and unions, leaving only reflexivity of literals:

REQUIREMENT 3.3 (LITERAL REFLEXIVITY)

$$\forall \ell.\ \exists r \in \mathcal{R}.\ \ell_r^{\leftarrow} = \ell = \ell_r^{\rightarrow} \ \wedge\ \forall \langle \tau^{\leftarrow}, \tau^{\rightarrow} \rangle \in \mathcal{P}_r.\ \tau^{\leftarrow} = \tau^{\rightarrow}$$

For our toy functional language, the reflexivity rule for the literal $\tau_i \to \tau_o$ is $\mathsf{Fun}(\tau_i, \tau_i, \tau_o, \tau_o)$.

3.3.4.2 *Rule Conversion*

Besides reflexivity, which we just discussed, and transitivity, which we will discuss next, there is a tight correspondence between the declarative rules and the reductive rules. The second step of the equivalence proof is to demonstrate this correspondence by converting each rule of one system into a combination of the rules of the other system.

For example, consider the reductive rule and its conversion (R-to-D) into declarative rules below:

$$\frac{\tau \leq \tau_i}{\tau \leq \tau_1 \cup \tau_2} \quad \overset{\text{R-to-D}}{\longmapsto} \quad \frac{\tau <: \tau_i \qquad \dfrac{}{\tau_i <: \tau_1 \cup \tau_2}}{\tau <: \tau_1 \cup \tau_2}$$

Note that whereas the reductive rule had one premise of the form $\tau \leq \tau_i$, the declarative proof has one assumption of the form $\tau <: \tau_i$.

To formalize this correspondence between premises in the rule and assumptions in the converted proof, Figure 3.3 defines proofs of declarative subtyping with assumptions.

Using this new definition, we can formalize the above conversion as follows:

$$\frac{\tau \leq \tau_i}{\tau \leq \tau_1 \cup \tau_2} \xmapsto{\text{R-to-D}} \frac{\dfrac{\langle \tau, \tau_i \rangle \in \{\langle \tau, \tau_i \rangle\}}{\{\langle \tau, \tau_i \rangle\} \vdash \tau <: \tau_i} \qquad \dfrac{}{\{\langle \tau, \tau_i \rangle\} \vdash \tau_i <: \tau_1 \cup \tau_2}}{\{\langle \tau, \tau_i \rangle\} \vdash \tau <: \tau_1 \cup \tau_2}$$

One can derive similar conversions for the remaining standard reductive rules as well. All that remains are the custom reductive literal rules.

Requirement 3.4 (Reductive-to-Declarative Literal Conversion)

$$\forall r \in \mathcal{R}.\ \ \mathcal{P}_r \vdash \ell_r^{\leftarrow} <: \ell_r^{\rightarrow}$$

For our functional language, the conversion for the rule $\mathsf{Fun}(\tau_i, \tau_i', \tau_o, \tau_o')$ is the following:

$$\frac{\tau_i' \leq \tau_i \qquad \tau_o \leq \tau_o'}{\tau_i \to \tau_o \leq \tau_i' \to \tau_o'} \xmapsto{\text{R-to-D}} \frac{\dfrac{\tau_i' <: \tau_i}{\tau_i \to \tau_o <: \tau_i' \to \tau_o} \qquad \dfrac{\tau_o <: \tau_o'}{\tau_i' \to \tau_o <: \tau_i' \to \tau_o'}}{\tau_i \to \tau_o <: \tau_i' \to \tau_o'}$$

These conversions enable us to inductively translate proofs of reductive subtyping into proofs of declarative subtyping. For the other direction, one needs the following, using proofs of reductive subtyping with assumptions as defined in Figure 3.3:

Requirement 3.5 (Declarative-to-Reductive Literal Conversion)

$$\forall r \in \mathcal{D}.\ \ \mathcal{P}_r \vdash \ell_r^{\leftarrow} \leq \ell_r^{\rightarrow}$$

Declarative Subtyping with Assumptions $\mathcal{P} \vdash \tau^{\leftarrow} <: \tau^{\rightarrow}$

$$\left(\begin{array}{c}\text{All rules in Figure 3.1,}\\ \text{replacing } \tau^{\leftarrow} <: \tau^{\rightarrow}\\ \text{with } \mathcal{P} \vdash \tau^{\leftarrow} <: \tau^{\rightarrow}\end{array}\right) \qquad \frac{\langle \tau^{\leftarrow}, \tau^{\rightarrow} \rangle \in \mathcal{P}}{\mathcal{P} \vdash \tau^{\leftarrow} <: \tau^{\rightarrow}}$$

Reductive Subtyping with Assumptions $\mathcal{P} \vdash \tau^{\leftarrow} \leq \tau^{\rightarrow}$

$$\left(\begin{array}{c}\text{All rules in Figure 3.2,}\\ \text{replacing } \tau^{\leftarrow} \leq \tau^{\rightarrow}\\ \text{with } \mathcal{P} \vdash \tau^{\leftarrow} \leq \tau^{\rightarrow}\end{array}\right) \qquad \frac{\langle \tau^{\leftarrow}, \tau^{\rightarrow} \rangle \in \mathcal{P}}{\mathcal{P} \vdash \tau^{\leftarrow} \leq \tau^{\rightarrow}}$$

Figure 3.3: Subtyping with Assumptions

With this and reflexivity, one can inductively translate proofs of declarative subtyping into reductive subtyping *provided* one can show reductive subtyping is transitive.

3.3.4.3 *Transitivity*

The last step to proving equivalence is transitivity, and also the most difficult step. Fortunately, one can once again repurpose well-foundedness to prove transitivity by reducing goals, just as was done for reflexivity. But whereas before goals were always of the form $\tau \leq \tau$, now they are of the form $\tau \leq \tau''$ where we have a proof of $\tau \leq \tau'$ and a proof of $\tau' \leq \tau''$ for some τ'. The key is to do a case analysis on the two proofs, taking advantage of the fact that τ' occurs in the conclusions of both proofs in order to eliminate a number of combinations of cases.

As an example, suppose the last steps of the two proofs are respectively as follows:

$$\textsc{Left}\ \frac{\tau \leq \tau_1' \qquad \tau \leq \tau_2'}{\tau \leq \tau_1' \cap \tau_2'} \qquad \frac{\tau_1' \leq \tau''}{\tau_1' \cap \tau_2' \leq \tau''}\ \textsc{Right}$$

Then this implies we have a proof of $\tau \leq \tau_1'$ and a proof of $\tau_1' \leq \tau''$. One can recursively reduce these two smaller proofs to get a proof of $\tau \leq \tau''$.

One can develop similar reductions for all the other possible cases in which either one of the last steps of the proofs is a standard reductive rule. And in every recursive reduction, the size of the proofs being reduced is strictly smaller. This ensures that reduction always eventually arrives at the case where both of the last steps are custom reductive literal rules. Here one can use well-foundedness to guarantee termination *provided* one can find a custom reductive literal rule to apply. Thus the largest step typically involved in proving equivalence is showing the following property.

Requirement 3.6 (Literal Transitivity) *For every pair of rules $r^{\leftarrow}$ and $r^{\rightarrow}$ in $\mathcal{R}$ such that $\ell^{\rightarrow}_{r^{\leftarrow}} = \ell^{\leftarrow}_{r^{\rightarrow}}$, there exists a rule r in $\mathcal{R}$ such that $\ell^{\leftarrow}_{r} = \ell^{\leftarrow}_{r^{\leftarrow}}$ and $\ell^{\rightarrow}_{r} = \ell^{\rightarrow}_{r^{\rightarrow}}$ and the following holds:*

$$\forall \langle \tau^{\leftarrow}, \tau^{\rightarrow} \rangle \in \mathcal{P}_r.\ \exists \tau.$$
$$(\mathcal{P}_{r^{\leftarrow}} \vdash \tau^{\leftarrow} \leq \tau\ \wedge\ \mathcal{P}_{r^{\rightarrow}} \vdash \tau \leq \tau^{\rightarrow})\ \ \vee\ \ (\mathcal{P}_{r^{\rightarrow}} \vdash \tau^{\leftarrow} \leq \tau\ \wedge\ \mathcal{P}_{r^{\leftarrow}} \vdash \tau \leq \tau^{\rightarrow})$$

The disjunction in the above requirement corresponds to the notion of variance in subtyping. Conceptually, a *covariant* premise of a rule is one in which the above will be proven using the left case of the disjunction, whereas a *contraviariant* premise uses the right case. The only difference is which assumptions the left-hand and right-hand subtyping proofs can make. In the covariant case, the proof of the left/right-hand subtyping can

assume the premises of the same-handed rule, whereas in the contravariant case the proof of the left/right-hand subtyping can assume the properties of the opposite-handed rule. This is best illustrated by our toy functional language.

Suppose $r^{\leftarrow}$ is $\mathsf{Fun}(\tau_i, \tau_i', \tau_o, \tau_o')$ and $r^{\rightarrow}$ is $\mathsf{Fun}(\tau_i', \tau_i'', \tau_o', \tau_o'')$, in combination concluding with $\tau_i \to \tau_o \leq \tau_i' \to \tau_o' \leq \tau_i'' \to \tau_o''$. Then r is the rule $\mathsf{Fun}(\tau_i, \tau_i'', \tau_o, \tau_o'')$, concluding with $\tau_i \to \tau_o \leq \tau_i'' \to \tau_o''$ as required. The following shows how the premises of r are proven with appropriate assumptions:

$$\frac{\langle \tau_i'', \tau_i' \rangle \in \mathcal{P}_{r^{\rightarrow}}}{\mathcal{P}_{r^{\rightarrow}} \vdash \tau_i'' \leq \tau_i'} \quad \wedge \quad \frac{\langle \tau_i', \tau_i \rangle \in \mathcal{P}_{r^{\leftarrow}}}{\mathcal{P}_{r^{\leftarrow}} \vdash \tau_i' \leq \tau_i} \qquad \frac{\langle \tau_o, \tau_o' \rangle \in \mathcal{P}_{r^{\leftarrow}}}{\mathcal{P}_{r^{\leftarrow}} \vdash \tau_o \leq \tau_o'} \quad \wedge \quad \frac{\langle \tau_o', \tau_o'' \rangle \in \mathcal{P}_{r^{\rightarrow}}}{\mathcal{P}_{r^{\rightarrow}} \vdash \tau_o' \leq \tau_o''}$$

Notice that the left-hand subtyping proof for *input* types uses the assumptions of the right-hand rule, whereas the left-hand subtyping proof for *output* types uses the assumptions of the left-hand rule. That is, the premise for input types is proven contravariantly, whereas the premise for output types is proven covariantly, which reflects the fact that function types are contravariant with respect to their input types and covariant with respect to their output types.

With this final requirement, one can prove that reductive subtyping is transitive. This was the final gap between declarative subtyping and reductive subtyping. Consequently, one finally knows that declarative subtyping and reductive subtyping are equivalent, and that proof search for reductive subtyping is a sound and complete algorithm for declarative subtyping. We formalize this general pattern in developing algorithms for subtyping systems with the following theorem.

Theorem 3.3.1 (Decidability of Declarative Subtyping)**.** *For every set of literals ℓ, set of declarative rules $\mathcal{D}$, and set of reductive rules $\mathcal{R}$ satisfying Requirements 3.1 through 3.6, proof search for reductive subtyping as defined in Figure 3.2 is a decision procedure for declarative subtyping as defined in Figure 3.1.*

Proof. Mechanically proved in Coq as outlined above [Muehlboeck and Tate, 2018a]. □

3.4 EMPOWERING UNIONS AND INTERSECTIONS

The previous section discussed typical decidable type systems with union and intersection types. Unfortunately, these typical systems fail to recognize many useful subtyping relations. Distributivity is one example, but more interesting are the relations specific to the particular literals at hand.

For example, consider our toy functional language. Suppose we have determined that a particular expression always produces a value of type τ_1 when given an integer. Suppose we have also determined that that same expression always produces a value of type τ_2 when given an integer. For the λ-calculus, Pierce recognized that, in this situation, that expression will always output values belonging to both τ_1 and τ_2 when given an integer. Thus, for Forsythe [Reynolds, 1988, 1997] he proposed adding the following axiom to the subtyping rules [Pierce, 1989]:

$$\frac{}{(\tau \to \tau_1') \cap (\tau \to \tau_2') <: \tau \to (\tau_1' \cap \tau_2')}$$

Extended Subtyping $\tau <:_{\mathcal{E}} \tau$

$$\begin{pmatrix}\text{All rules in Figure 3.1,}\\ \text{replacing } \tau^{\leftarrow} <: \tau^{\rightarrow}\\ \text{with } \tau^{\leftarrow} <:_{\mathcal{E}} \tau^{\rightarrow}\end{pmatrix} \qquad \frac{}{\tau \cap (\tau_1 \cup \tau_2) <:_{\mathcal{E}} (\tau \cap \tau_1) \cup (\tau \cap \tau_2)} \qquad \frac{\langle \tau^{\leftarrow}, \tau^{\rightarrow} \rangle \in \mathcal{E}}{\tau^{\leftarrow} <:_{\mathcal{E}} \tau^{\rightarrow}}$$

Figure 3.4: Extended Subtyping

τ	$\mathrm{DNF}_c(\tau)$ where $c : \vec{\ell} \to \tau$
$\bot$	$\bot$
$\tau_1 \cup \tau_2$	$\mathrm{DNF}_c(\tau_1) \cup \mathrm{DNF}_c(\tau_2)$
$\top$	$c([])$
$\tau_1 \cap \tau_2$	$\mathrm{DNF}_{\lambda \vec{\ell}_1.\, \mathrm{DNF}_{\lambda \vec{\ell}_2.c(\vec{\ell}_1 +\!\!+ \vec{\ell}_2)}(\tau_2)}(\tau_1)$
ℓ	$c([\ell])$

Figure 3.5: Definition of DNF_c

The goal of our framework is to develop decision procedures for declarative subtyping systems empowered with such axioms (and distributivity). That is, given a set $\mathcal{E} \subseteq \tau \times \tau$, we aim to provide a decision procedure for *extended* subtyping as defined in Figure 3.4. Furthermore, since the declarative subtyping system being extended had already been proven decidable, we want to reuse as much of that algorithm and proof work as possible, only requiring the designer to do the work specific to the extensions at hand. Since distributivity is an extension of arbitrary union and intersection types, we tackle that extension first. Afterwards, we will consider extensions incorporating literals, such as the extension $\mathcal{E}_{\mathsf{Out}}$ defined as $\{\langle (\tau \to \tau_1') \cap (\tau \to \tau_2'), \tau \to (\tau_1' \cap \tau_2') \rangle \mid \tau, \tau_1', \tau_2'\}$, which corresponds to the above extension to Forsythe proposed by Pierce.

3.4.1 *Distributivity*

Observe that the right-hand type of the distributivity axiom in Figure 3.4 is simply the result of distributing the intersection in the left-hand type over the union in the left-hand type. That is, there is an operation we can apply to the left-hand type to arrive at the right-hand type. The high-level strategy of our *integrated*-subtyping technique is to transform the left-hand type in order to *integrate* the axiom into the type itself. In the case of distributivity, this integrating operator simply maps a type to its disjunctive normal form, effectively distributing all its intersections over its unions. Prior works have employed a similar strategy [Aiken and Wimmers, 1993; Ancona and Corradi, 2016; Frisch, Castagna, and Véronique Benzaken, 2008; Peyton Jones et al., 2007; Pierce, 1991; Reynolds, 1988], but here we demonstrate that this can be employed in both a principled and extensible manner.

We formalize this operator in Figure 3.5 using a continuation-style implementation. The intermediate continuations effectively collect the literals to be intersected together, calling the original continuation c after all the literals have been collected. By instantiating c with the operator $\bigcap$ that simply intersects the literals together, $\mathrm{DNF}_\cap$ maps each type to its disjunctive normal form.

The operator $\mathrm{DNF}_\cap$ exhibits three important properties. First, $\mathrm{DNF}_\cap(\tau \cap (\tau_1 \cup \tau_2))$ is declaratively/reductively a subtype of $(\tau \cap \tau_1) \cup (\tau \cap \tau_2)$. This illustrates that $\mathrm{DNF}_\cap$ integrates the distributivity axiom into the left-hand type, which will ensure that integrated subtyping is distributive. Second, $\mathrm{DNF}_\cap(\tau)$ is declaratively/reductively a subtype of τ, which will

ensure that integrated subtyping is still reflexive. Third, whenever τ is in disjunctive normal form, say by being in the image of $\mathrm{DNF}_\cap$, one can show that τ is declaratively/reductively a subtype of $\mathrm{DNF}_\cap(\tau')$ whenever τ is declaratively/reductively a subtype of τ', which will ensure that integrated subtyping is still transitive. As a consequence of these properties, integrated subtyping will be equivalent to extended subtyping. But before we properly define integrated subtyping, we want to consider how to integrate extensions beyond distributivity as well.

3.4.2 *Intersectors*

As we distribute intersections over unions, we have the opportunity to do more with the resulting union of intersections. In particular, DNF accepts any operator mapping a list of literals to a type. Instead of simply intersecting the literals together, this operator could incorporate domain-specific reasoning about the literals. For example, the operator could check whether two literals in the list are disjoint classes, in which case it could simply return $\bot$. This provides a means to integrate more extensions into the type as we transform it.

We call such an operator an *intersector*, and we denote intersectors using $\bigcap$. One simple case is where the intersector is simply $\bigcap$, which has the effect of only adding distributivity. But since the truly interesting expressivity comes from the domain-specific extensions $\mathcal{E}$, we use the following requirement to formalize the intent that the intersector $\bigcap$ combines with DNF to form an *integrator* $\mathrm{DNF}_\sqcap$ that integrates those extensions into the types.

REQUIREMENT 3.7 (INTERSECTOR COMPLETENESS) $\forall \langle \tau^{\leftarrow}, \tau^{\rightarrow} \rangle \in \mathcal{E}.\ \mathrm{DNF}_{\sqcap}(\tau^{\leftarrow}) <: \tau^{\rightarrow}$

One easy way to integrate all extensions into an intersector is simply to define the intersector so that it always outputs $\bot$. Such an intersector is clearly undesirable because it conceptually is doing too much; that is, it is integrating more information into the type than can be deduced from the extensions. As such, we also impose the following requirement that ensures the intersector integrates nothing more than what the extensions can deduce.

REQUIREMENT 3.8 (INTERSECTOR SOUNDNESS) $\forall \vec{\ell}.\ \bigcap \vec{\ell} <:_{\mathcal{E}} \bigsqcap \vec{\ell}$

As an example, consider the following intersector $\bigsqcap_{\mathsf{Out}}$ for our running Forsythe extension $\mathcal{E}_{\mathsf{Out}}$:

$$\bigsqcap_{\mathsf{Out}} \vec{\ell} = \bigcap_{\varnothing \subset \mathcal{L} \subseteq \vec{\ell}} \left(\bigcap_{(\tau_i \to \tau_o) \in \mathcal{L}} \tau_i \right) \to \left(\bigcap_{(\tau_i \to \tau_o) \in \mathcal{L}} \tau_o \right)$$

To see why this works, consider the case in which there are two function-type literals $\tau_i \to \tau_o$ and $\tau_i' \to \tau_o'$. In this case, the result of $\bigsqcap_{\mathsf{Out}}$ is $(\tau_i \to \tau_o) \cap (\tau_i' \to \tau_o') \cap (\tau_i \cap \tau_i' \to \tau_o \cap \tau_o')$. If τ_i equals τ_i' so that we can refer to both as, say, τ, then the last component of this intersection is clearly equivalent to $\tau \to (\tau_o \cap \tau_o')$, thereby integrating the extension $\mathcal{E}_{\mathsf{Out}}$. Note that variance of function types makes $\bigsqcap_{\mathsf{Out}}$ still sound with respect to $\mathcal{E}_{\mathsf{Out}}$ even when τ_i does not equal τ_i'.

Integrated Subtyping $\tau \leq_{\sqcap} \tau \qquad \tau \leq^{\sqcap} \tau$

$$\frac{\mathrm{DNF}_{\sqcap}(\tau) \leq^{\sqcap} \tau'}{\tau \leq_{\sqcap} \tau'} \qquad \frac{\left(\begin{array}{c}\text{All rules in Figure 3.2 except literal subtyping,}\\ \text{replacing } \tau^{\leftarrow} \leq \tau^{\rightarrow} \text{ with } \tau^{\leftarrow} \leq^{\sqcap} \tau^{\rightarrow}\end{array}\right) \quad r \in \mathcal{R} \qquad \forall \langle \tau^{\leftarrow}, \tau^{\rightarrow} \rangle \in \mathcal{P}_r.\ \ \mathrm{DNF}_{\sqcap}(\tau^{\leftarrow}) \leq^{\sqcap} \tau^{\rightarrow}}{\ell_r^{\leftarrow} \leq^{\sqcap} \ell_r^{\rightarrow}}$$

Figure 3.6: Integrated Subtyping

3.4.3 *Integrated Subtyping*

Now that we have an intersector of literals $\bigcap$ that corresponds to the intended extension $\mathcal{E}$, we define our integrated subtyping rules in Figure 3.6, incorporating the intersector by *integrating* left-hand types using $\mathrm{DNF}_{\sqcap}$. The integrated subtyping rules essentially describe the same algorithm as the reductive subtyping rules with just two small changes. The first is that integrated subtyping $\leq_{\sqcap}$ starts by integrating the left-hand type and then calling the recursive procedure described by $\leq^{\sqcap}$. The second is that, when this procedure recurses in the case of literals, it again integrates the left-hand type. The effect of this is that the left-hand type is always *integrated* at critical points where the procedure recurses, meaning it is essentially in the image of $\mathrm{DNF}_{\sqcap}$.

Lemma 1 (Integrated Soundness) *Assuming hitherto requirements:*

$$\forall \tau^{\leftarrow}, \tau^{\rightarrow}.\ \ \tau^{\leftarrow} \leq_{\sqcap} \tau^{\rightarrow} \implies \tau^{\leftarrow} <:_{\mathcal{E}} \tau^{\rightarrow}$$

Soundness of integrated subtyping with respect to extended subtyping follows from simply adapting the proof of soundness of reductive typing to incorporate Requirement 3.8 whenever integrated subtyping integrates the

left-hand type. Completeness, on the other hand, is much more challenging to achieve. Requirement 3.7 is just one of many steps to this effect. In the following, we first ensure that integrated subtyping is still decidable, and then the remainder of the section establishes completeness so that we can use the decision procedure for integrated subtyping as a decision procedure for extended subtyping. As our running example, we will demonstrate that these steps produce a decision procedure for our toy functional language extended with $\mathcal{E}_{\mathsf{Out}}$.

3.4.4 *Decidability*

Integrated subtyping is syntax-directed for the same reason as reductive subtyping. Thus we can decide integrated subtyping by recursively searching the proof space *provided* we can guarantee its recursion terminates. We already have a well-founded measure for reductive subtyping, and we can repurpose it to prove termination of integrated subtyping provided the intersector preserves it.

Requirement 3.9 (Measure Preservation) $\forall \tau^{\leftarrow}, \tau^{\rightarrow}.\ \ m(\mathrm{DNF}_{\sqcap}(\tau^{\leftarrow}), \tau^{\rightarrow}) \preceq m(\tau^{\leftarrow}, \tau^{\rightarrow})$

In the common case where m is defined on unions and intersections using joins, this amounts to only ensuring that $m(\bigsqcap \vec{\ell}, \tau^{\rightarrow})$ is always less than or equal to $m(\bigcap \vec{\ell}, \tau^{\rightarrow})$. For $\bigsqcap_{\mathsf{Out}}$ this holds because the measures of both sides are in fact always equal. That is, $\bigsqcap_{\mathsf{Out}}$ preserves termination for our toy functional language because it does not increase the nesting depth of function types.

LEMMA 2 (INTEGRATED DECIDABILITY) *Assuming hitherto requirements, the relation $\leq_{\sqcap}$ is decidable.*

3.4.5 *Integrating*

Clearly the literal intersector $\bigsqcap$ and the corresponding type integrator $\mathrm{DNF}_{\sqcap}$ are the key new components of integrated subtyping. The high-level intuition of how they work is that the integrator $\mathrm{DNF}_{\sqcap}$ forms a comonad on subtyping. Integrated subtyping then is *conceptually* akin to the Kleisli category of this comonad. However, integrated subtyping *recursively* integrates the comonad into its rules and its corresponding proof-search algorithm. As such, we cannot simply reuse standard theorems from category theory to achieve our results, and much of the challenge lies in addressing this recursive integration of $\mathrm{DNF}_{\sqcap}$ into the subtyping system. But we can still use these theorems as inspiration in establishing the equivalence of integrated and extended subtyping.

3.4.5.1 *Dereliction*

Intuitively, the intersector $\bigsqcap$ makes the type more precise. That is, $\bigsqcap$ should add information to the type, not remove information, as formalized by the following requirement.

REQUIREMENT 3.10 (LITERAL DERELICTION) $\forall \vec{\ell}.\ \bigsqcap \vec{\ell} \leq \bigcap \vec{\ell}$

This trivially holds for $\bigsqcap_{\mathsf{Out}}$ since the resulting intersection always contains the input literals.

τ	$\mathrm{dnf}_\phi(\tau)$ where $\phi \subseteq \vec{\ell}$	$\mathrm{dnf}^{\cap}_\phi(\tau)$ where $\phi \subseteq \vec{\ell}$
$\bot$	**true**	**false**
$\tau_1 \cup \tau_2$	$\mathrm{dnf}_\phi(\tau_1) \wedge \mathrm{dnf}_\phi(\tau_2)$	**false**
$\top$	$\mathrm{dnf}^{\cap}_\phi(\top)$	$\phi([])$
$\tau_1 \cap \tau_2$	$\mathrm{dnf}^{\cap}_\phi(\tau_1 \cap \tau_2)$	$\mathrm{dnf}^{\cap}_{\lambda\vec{\ell}_1.\,\mathrm{dnf}^{\cap}_{\lambda\vec{\ell}_2.\phi(\vec{\ell}_1 +\!\!+ \vec{\ell}_2)}(\tau_2)}(\tau_1)$
ℓ	$\mathrm{dnf}^{\cap}_\phi(\ell)$	$\phi([\ell])$

Figure 3.7: Definition of dnf_ϕ

LEMMA 3 (DERELICTION) *Assuming hitherto requirements:*

$$\forall \tau^{\leftarrow}, \tau^{\rightarrow}.\ \tau^{\leftarrow} \leq \tau^{\rightarrow} \implies \mathrm{DNF}_{\sqcap}(\tau^{\leftarrow}) \leq \tau^{\rightarrow}$$

3.4.5.2 *Intersected*

Just like $\mathrm{DNF}_\cap$ produces types in disjunctive normal form, we need some property that describes the key trait of the image of the intersector $\bigsqcap$. Let ϕ be some such property of lists of literals. We say a list of literals is *intersected* if it satisfies ϕ. We extend this predicate to arbitrary types using the dnf_ϕ predicate defined in Figure 3.7, which informally states that a type is *integrated* if it is a union of intersections of *intersected* lists of literals.

Just like the goal of $\mathrm{DNF}_\cap$ is to produce types that are in disjunctive normal form, the goal of $\mathrm{DNF}_\sqcap$ is to produce integrated types. This is ensured by the following requirement.

REQUIREMENT 3.11 (INTERSECTOR INTEGRATED) $\forall \vec{\ell}.\ \ \mathrm{dnf}_\phi(\bigsqcap \vec{\ell})$

For our running example, we can define ϕ_{Out} to hold for a list of function-type literals when, given any two function types $\tau_1 \to \tau_1'$ and $\tau_2 \to \tau_2'$ in the list, the combined function-type literal $(\tau_1 \cap \tau_2) \to (\tau_1' \cap \tau_2')$ is also in the list up to syntactic equivalence, meaning the only differences are

resolved by associativity, commutativity, and idempotence of intersection types.

LEMMA 4 (INTEGRATOR INTEGRATED) *Assuming hitherto requirements:*

$$\forall \tau.\ \mathrm{dnf}_{\phi}(\mathrm{DNF}_{\sqcap}(\tau))$$

3.4.5.3 *Promotion*

For our purposes, the critical property of disjunctive normal form is that, whenever a type τ is in disjunctive normal form and is a declarative/reductive subtype of some other type τ', then τ is furthermore a declarative/reductive subtype of $\mathrm{DNF}_{\cap}(\tau')$. This property further substantiates the intuition that disjunctive normal form and $\mathrm{DNF}_{\cap}$ integrate distributivity directly into types. We want there to be a similar relationship between integrated types and the integrator $\mathrm{DNF}_{\sqcap}$. This relationship should extend from a similar relationship between intersected literals and the intersector $\bigsqcap$. The following requirement formalizes that relationship.

REQUIREMENT 3.12 (LITERAL PROMOTION) *For every two lists of literals $\vec{\ell}$ and $\vec{\ell}'$ and list of reductive literal subtyping rules $\vec{r} \subseteq \mathcal{R}$ such that*

$$\forall \ell' \in \vec{\ell}'.\ \exists r \in \vec{r}.\ \ell_r^{\rightarrow} = \ell' \qquad \wedge \qquad \forall r \in \vec{r}.\ \ell_r^{\leftarrow} \in \vec{\ell}$$

if $\phi(\vec{\ell})$ holds then there exists a list of reductive literal subtyping rules $\vec{r}_{\sqcap} \subseteq \mathcal{R}$ such that $\bigsqcap \vec{\ell}'$ is of the form $\cdots \cup \left(\bigcap_{r_{\sqcap} \in \vec{r}_{\sqcap}} \ell_{r_{\sqcap}}^{\rightarrow}\right) \cup \ldots$ and for all reductive literal subtyping rules $r_{\sqcap}$ in $\vec{r}_{\sqcap}$

$$\ell_{r_{\sqcap}}^{\leftarrow} \in \vec{\ell} \qquad \wedge \qquad \forall \langle \tau^{\leftarrow}, \tau^{\rightarrow} \rangle \in \mathcal{P}_{r_{\sqcap}}.\ \left(\bigcup_{r \in \vec{r}} \mathcal{P}_r\right) \vdash \tau^{\leftarrow} \leq \tau^{\rightarrow}$$

The premise of Requirement 3.12 indicates we have some list of rules ($\vec{r}$) that can be used to prove that one intersection of literals ($\bigcap \vec{\ell}$) is a subtype of another ($\bigcap \vec{\ell}'$). The requirement, then, is essentially that if $\vec{\ell}$ is furthermore *intersected* (i.e. $\phi(\vec{\ell})$ holds) then it should be possible to prove from the combined premises of $\vec{r}$ that $\bigcap \vec{\ell}$ is furthermore a subtype of $\bigsqcap \vec{\ell}'$. More specifically, the requirement is that there is a list of rules ($\vec{r}_\sqcap$) that proves that $\bigcap \vec{\ell}$ is a subtype of $\bigsqcap \vec{\ell}'$ and whose premises can be proven from the premises of $\vec{r}$. In a sense, Requirement 3.12 states that there is a proof-theoretic analog to intersectors $\bigsqcap$ in that the list of rules $\vec{r}_\sqcap$ is the "promoted" analog of $\vec{r}$.

For our running example, suppose the list of rules $\vec{r}$ consists of the rules r_1 and r_2, which are $\mathsf{Fun}(\tau_i^1, \tau_i^{1\prime}, \tau_o^1, \tau_o^{1\prime})$ and $\mathsf{Fun}(\tau_i^2, \tau_i^{2\prime}, \tau_o^2, \tau_o^{2\prime})$ respectively. This means the list of literals $\vec{\ell}'$ is $\tau_i^{1\prime} \to \tau_o^{1\prime}$ and $\tau_i^{2\prime} \to \tau_o^{2\prime}$, and it means the list of literals $\vec{\ell}$ contains at least $\tau_i^1 \to \tau_o^1$ and $\tau_i^2 \to \tau_o^2$. The result of applying the intersector $\bigsqcap_{\mathsf{Out}}$ to $\vec{\ell}'$ is then

$$(\tau_i^{1\prime} \to \tau_o^{1\prime}) \cap (\tau_i^{2\prime} \to \tau_o^{2\prime}) \cap (\tau_i^{1\prime} \cap \tau_i^{2\prime} \to \tau_o^{1\prime} \cap \tau_o^{2\prime})$$

This intersection is comprised of three literals, which means we need three "promoted" rules $\vec{r}_\sqcap$. In this case the first two rules are simply r_1 and r_2, whose premises are trivially provable from the premises of $\vec{r}$. For the last rule, we need a rule that proves that some literal in $\vec{\ell}$ is a subtype of $\tau_i^{1\prime} \cap \tau_i^{2\prime} \to \tau_o^{1\prime} \cap \tau_o^{2\prime}$. In general, such a rule does not necessarily exist, but here we are allowed to assume that $\vec{\ell}$ is *intersected*, meaning it satisfies ϕ_{Out}. Since we know $\vec{\ell}$ contains at least $\tau_i^1 \to \tau_o^1$ and $\tau_i^2 \to \tau_o^2$, by the definition of ϕ_{Out} we know $\vec{\ell}$ must also contain the literal $(\tau_i^1 \cap \tau_i^2) \to (\tau_o^1 \cap \tau_o^2)$ (or something syntactically equivalent). Thus we

can use $\mathsf{Fun}(\tau_i^1 \cap \tau_i^2, \tau_i^{1\prime} \cap \tau_i^{2\prime}, \tau_o^1 \cap \tau_o^2, \tau_o^{1\prime} \cap \tau_o^{2\prime})$ (or something syntactically equivalent) as our final "promoted" rule, since the fact that $\tau_i^{1\prime} \cap \tau_i^{2\prime}$ is a subtype of $\tau_i^1 \cap \tau_i^2$ and that $\tau_o^1 \cap \tau_o^2$ is a subtype of $\tau_o^{1\prime} \cap \tau_o^{2\prime}$ is easily proven from the *combined* premises of $\vec{r}$ (which are $\langle \tau_i^{1\prime}, \tau_i^1 \rangle$, $\langle \tau_o^1, \tau_o^{1\prime} \rangle$, $\langle \tau_i^{2\prime}, \tau_i^2 \rangle$, and $\langle \tau_o^2, \tau_o^{2\prime} \rangle$).

LEMMA 5 (PROMOTION) *Assuming hitherto requirements:*

$$\forall \tau^{\leftarrow}, \tau^{\rightarrow}.\ \ \mathrm{dnf}_{\phi}(\tau^{\leftarrow}) \ \ \wedge \ \ \tau^{\leftarrow} \leq \tau^{\rightarrow} \implies \tau^{\leftarrow} \leq \mathrm{DNF}_{\sqcap}(\tau^{\rightarrow})$$

Promotion for reductive subtyping is easily proven by induction. However, promotion for integrated subtyping is not nearly so simple. The problem is that literal promotion makes use of reductive proofs with assumptions. While such proofs can easily be converted into reductive subtyping (without assumptions) when reductive subtyping holds for each of the assumptions, the same is not true for integrated subtyping because its subtyping rule for literals differs. In order to bridge this difference, one first needs to prove that integrated subtyping is monotonic with respect to reductive subtyping, and the proof of this involves a complex mix of transitivity reduction, promotion of reductive subtyping, and "big-step" well-founded coinduction alternating with "small-step" induction. Fortunately, our framework abstracts away all this complex proof machinery from the concerns of its users, providing the following convenient lemmas.

LEMMA 6 (INTEGRATED MONOTONICITY) *Assuming hitherto requirements:*

$$\forall \tau_1, \tau_2, \tau_3, \tau_4.\ \ \tau_1 \leq \tau_2 \ \ \wedge \ \ \tau_2 \leq_{\sqcap} \tau_3 \ \ \wedge \ \ \tau_3 \leq \tau_4 \implies \tau_1 \leq_{\sqcap} \tau_4$$

LEMMA 7 (INTEGRATED ASSUMPTIONS) *Assuming hitherto requirements:*

$$\forall \mathcal{P}, \tau^{\leftarrow}, \tau^{\rightarrow}.\ \mathcal{P} \vdash \tau^{\leftarrow} \leq \tau^{\rightarrow} \implies \left(\forall \langle \tau_p^{\leftarrow}, \tau_p^{\rightarrow} \rangle \in \mathcal{P}.\ \tau_p^{\leftarrow} \leq_{\sqcap} \tau_p^{\rightarrow}\right) \implies \tau^{\leftarrow} \leq_{\sqcap} \tau^{\rightarrow}$$

LEMMA 8 (INTEGRATED PROMOTION) *Assuming hitherto requirements:*

$$\forall \tau^{\leftarrow}, \tau^{\rightarrow}.\ \tau^{\leftarrow} \leq_{\sqcap} \tau^{\rightarrow} \implies \tau^{\leftarrow} \leq_{\sqcap} \mathrm{DNF}_{\sqcap}(\tau^{\rightarrow})$$

3.4.6 *Equivalence of Extended and Integrated Subtyping*

We have now established the essential comonad-like properties of $\mathrm{DNF}_{\sqcap}$. Now we move on to utilizing these properties to prove that integrated subtyping admits the rules of extended subtyping that were removed to achieve decidability.

3.4.6.1 *Reflexivity*

Reflexivity is a corollary of the prior lemmas. In particular, a reductive proof with assumptions in which the assumption set $\mathcal{P}$ is empty directly corresponds to reductive subtyping, so Lemma 7 indicates that reductive subtyping implies integrated subtyping. As a consequence, reflexivity of reductive subtyping implies reflexivity of integrated subtyping.

LEMMA 9 (INTEGRATED REFLEXIVITY) *Assuming hitherto requirements:*

$$\forall \tau.\ \tau \leq_{\sqcap} \tau$$

3.4.6.2 *Rule Conversion*

Integrated subtyping relies upon the reductive literal subtyping rules $\mathcal{R}$, whereas extended subtyping relies upon the declarative literal subtyping rules $\mathcal{D}$. For our proof of equivalence of declarative and reductive subtyping, we already required a conversion of declarative literal rules into reductive subtyping proofs with assumptions (Requirement 3.5), and for our proof of integrated promotion, we already demonstrated that reductive proofs with assumptions can be converted into integrated subtypings under appropriate assumptions (Lemma 7). As a consequence, we can simply reuse the required conversions of literal rules to prove that integrated subtyping also admits the declarative literal rules.

Lemma 10 (Declarative-to-Integrated Literal Conversion) *Assuming hitherto requirements:*

$$\forall r \in \mathcal{D}.\ \ (\forall \langle \tau^{\leftarrow}, \tau^{\rightarrow} \rangle \in \mathcal{P}_r.\ \ \tau^{\leftarrow} \leq_{\sqcap} \tau^{\rightarrow}) \implies \ell_r^{\leftarrow} \leq_{\sqcap} \ell_r^{\rightarrow}$$

3.4.6.3 *Transitivity*

The final step is to prove transitivity of subtyping. As with reductive subtyping, conceptually this is done by recursively reducing the left-hand proof and right-hand proof, eventually making progress towards a combined subtyping proof, which in turn ensures termination due to well-foundedness of integrated subtyping. However, there is a key difference here that arises when reducing applications of literal rules, which we illustrate using our running example.

Suppose the left-hand proof is an application of $\mathsf{Fun}(\tau_i, \tau_i', \tau_o, \tau_o')$ and the right-hand proof is an application of $\mathsf{Fun}(\tau_i', \tau_i'', \tau_o', \tau_o'')$, meaning the integrated proofs being reduced appear as follows:

$$\dfrac{\dfrac{\vdots}{\mathrm{DNF}_\sqcap(\tau_i') \leq^\sqcap \tau_i} \qquad \dfrac{\vdots}{\mathrm{DNF}_\sqcap(\tau_o) \leq^\sqcap \tau_o'}}{\tau_i \to \tau_i' \leq^\sqcap \tau_o \to \tau_o'}$$

$$\dfrac{\dfrac{\vdots}{\mathrm{DNF}_\sqcap(\tau_i'') \leq^\sqcap \tau_i'} \qquad \dfrac{\vdots}{\mathrm{DNF}_\sqcap(\tau_o') \leq^\sqcap \tau_o''}}{\tau_i' \to \tau_i'' \leq^\sqcap \tau_o' \to \tau_o''}$$

Notice there is a mismatch between the conclusions of the subproofs: the left-hand types are *integrated* but the right-hand types are not. But we can resolve this mismatch by *promoting* the appropriate proofs so that we can recursively reduce the transitive pairs of proofs for i and for o. Thus we can make progress towards proving $\tau_i \to \tau_i'' \leq^\sqcap \tau_o \to \tau_o''$ by applying $\mathsf{Fun}(\tau_i, \tau_i'', \tau_o, \tau_o'')$.

While this intuition is the essence of how transitivity is proven, the actual proof once again involves a complex mix of transitivity reduction, promotion of integrated subtyping, and "big-step" well-founded coinduction alternating with "small-step" induction. In particular, integrated subtyping is not itself reductive—transitivity is not simply achieved by showing that each cutpoint reduces. While that is a step of the proof, the much more challenging step is recursively incorporating the integrator into transitivity, with promotion essentially integrating the intended axioms into the subtyping proofs themselves just like the integrator does

with types. Thus integrated subtyping is proof-theoretically more powerful than reductive subtyping, making it likely *necessary* for these more advanced subtyping systems. Fortunately, our framework abstracts away this complex proof machinery, providing the following convenient lemmas and the main theorem of this chapter.

LEMMA 11 (INTEGRATED TRANSITIVITY) *Assuming hitherto requirements:*

$$\forall \tau_1, \tau_2, \tau_3.\ \ \tau_1 \leq_\sqcap \tau_2 \ \wedge\ \tau_2 \leq_\sqcap \tau_3 \implies \tau_1 \leq_\sqcap \tau_3$$

LEMMA 12 (INTEGRATED COMPLETENESS) *Assuming hitherto requirements:*

$$\forall \tau^{\leftarrow}, \tau^{\rightarrow}.\ \ \tau^{\leftarrow} <:_{\mathcal{E}} \tau^{\rightarrow} \implies \tau^{\leftarrow} \leq_\sqcap \tau^{\rightarrow}$$

Theorem 3.4.1 (Decidability of Extended Subtyping)**.** *For every set of literals ℓ, set of declarative rules $\mathcal{D}$, set of reductive rules $\mathcal{R}$, set of extensions $\mathcal{E}$, intersector of literals $\sqcap$, and intersected predicate ϕ satisfying Requirements 3.1 through 3.12, proof search for integrated subtyping as defined in Figure 3.6 is a decision procedure for extended subtyping as defined in Figure 3.4.*

Proof. Mechanically proved in Coq [Muehlboeck and Tate, 2018a], along with a proof that the proof search can be safely optimized by using a rule "prioritization". This means that when proof search encounters a situation in which multiple rules are applicable, it does not need to search the rules that have lower "priority" than some other applicable rule. In this case, the high-priority rules are the left union and bottom rules, the mid-priority rules are all the right rules, and the low-priority rules are the left intersection rules and the custom literal rules. □

Thus, since we already illustrated that $\bigsqcap_{\mathsf{Out}}$ satisfies the requirements with respect to $\mathcal{E}_{\mathsf{Out}}$, this means proof search for $\leq_{\bigsqcap_{\mathsf{Out}}}$ is a decision procedure for $<:_{\mathcal{E}_{\mathsf{Out}}}$. But $\mathcal{E}_{\mathsf{Out}}$ is only one of the extensions Pierce considered for Forsythe, one that he already proved decidable (without union types) [Pierce, 1989]. Rather than redo this work each time one considers another extension, next we discuss how one can develop each extension separately and then easily compose the extensions together with just one more simple proof. We will be illustrating composability with our main application, Ceylon, rather than our toy functional language.

3.5 COMPOSABILITY

So far, we have had only a relatively simple integrator as an example. As we discussed in Section 3.2, Ceylon uses union and intersection types to build several type-system features, which requires more involved type integrators. We would like to be able to specify them separately and then only later compose them into a single master type integrator. In this section, we discuss how to do this.

Suppose one has developed two intersectors $\bigsqcap_1$ and $\bigsqcap_2$ and respective intersected predicates ϕ_1 and ϕ_2 on the same set of literals ℓ and literal subtyping rules $\mathcal{R}$. One can compose them as follows:

$$\bigsqcap \vec{\ell} ::= \mathrm{DNF}_{\sqcap_2}(\bigsqcap_1 \vec{\ell}) \qquad\qquad \phi(\vec{\ell}) ::= \phi_1(\vec{\ell}) \wedge \phi_2(\vec{\ell})$$

The only thing one needs to show for this composition to work as expected is that $\bigsqcap_2$ preserves ϕ_1.

REQUIREMENT 3.13 (INTERSECTED PRESERVATION) $\forall \vec{\ell}.\ \ \phi_1(\vec{\ell}) \Rightarrow \mathrm{dnf}_{\phi_1}(\bigsqcap_2 \vec{\ell})$

Theorem 3.5.1 (Intersector Composability). *For every set of literals ℓ, set of declarative rules $\mathcal{D}$, set of reductive rules $\mathcal{R}$, and sets of extensions $\mathcal{E}_1$ and $\mathcal{E}_2$ with corresponding intersectors of literals $\bigsqcap_1$ and $\bigsqcap_2$ and intersected predicates ϕ_1 and ϕ_2 each satisfying Requirements 3.1 through 3.12, satisfying Requirement 3.13 implies the composed intersector $\bigsqcap \vec{\ell} = \mathrm{DNF}_{\sqcap_2}(\bigsqcap_1 \vec{\ell})$ and intersected predicate $\phi(\vec{\ell}) = \phi_1(\vec{\ell}) \wedge \phi_2(\vec{\ell})$ satisfy Requirements 3.1 through 3.12 with respect to the extension $\mathcal{E}_1 \cup \mathcal{E}_2$.*

Proof. Mechanically proved in Coq as outlined above [Muehlboeck and Tate, 2018a]. □

3.6 APPLICATION TO CEYLON

We have been using a toy language as our running illustrative example. In this section, we will show how to use integrated subtyping to uniformly implement various aspects of an industry language, namely the Ceylon language [King, 2013] and its features that we discussed in Section 3.2.

3.6.1 *Unempowered Ceylon*

Ceylon is an object-oriented language, and its literals are class types. More specifically, Ceylon uses generics with declaration-site variance, so its class types are parameterized with some number of arguments, some of which are covariant and some of which are contravariant. For the sake of concision, we approximate this expressiveness with single-parameter class types employing *use*-site variance. As such, our literals will be of the form $C\langle \mathbf{in}\ \tau_i\ \mathbf{out}\ \tau_o \rangle$, with the **in** argument being the *contravariant*

Ceylon Subtyping $\tau <: \tau \qquad \tau \leq \tau \qquad C\langle\cdot\rangle <:: C\langle\dot{\tau}\rangle$

$$\frac{\tau_i' <: \tau_i \qquad \tau_o <: \tau_o'}{C\langle \mathbf{in}\ \tau_i\ \mathbf{out}\ \tau_o\rangle <: C\langle \mathbf{in}\ \tau_i'\ \mathbf{out}\ \tau_o'\rangle}$$

$$\frac{C\langle\cdot\rangle <:: C'\langle\dot{\tau}\rangle}{C\langle \mathbf{in}\ \tau_i\ \mathbf{out}\ \tau_o\rangle <: C'\langle \mathbf{in}\ \dot{\tau}[\mathbf{in}\ \tau_o\ \mathbf{out}\ \tau_i]\ \mathbf{out}\ \dot{\tau}[\mathbf{in}\ \tau_i\ \mathbf{out}\ \tau_o]\rangle}$$

$$\frac{}{C\langle\cdot\rangle \leq:: C\langle\cdot\rangle} \qquad \frac{C\langle\cdot\rangle <:: C'\langle\dot{\tau}\rangle \quad C'\langle\cdot\rangle \leq:: C''\langle\dot{\tau}'\rangle}{C\langle\cdot\rangle \leq:: C''\langle\dot{\tau}'[\dot{\tau}]\rangle} \qquad \frac{C\langle\cdot\rangle \leq:: C'\langle\dot{\tau}\rangle \quad \tau_i' \leq \dot{\tau}[\mathbf{in}\ \tau_o\ \mathbf{out}\ \tau_i] \quad \dot{\tau}[\mathbf{in}\ \tau_i\ \mathbf{out}\ \tau_o] \leq \tau_o'}{C\langle \mathbf{in}\ \tau_i\ \mathbf{out}\ \tau_o\rangle \leq C'\langle \mathbf{in}\ \tau_i'\ \mathbf{out}\ \tau_o'\rangle}$$

$\dot{\tau}$	$\dot{\tau}[\mathbf{in}\ \tau_i\ \mathbf{out}\ \tau_o]$
$\cdot$	τ_o
$\bot$	$\bot$
$\dot{\tau}_1 \cup \dot{\tau}_2$	$\dot{\tau}_1[\mathbf{in}\ \tau_i\ \mathbf{out}\ \tau_o] \cup \dot{\tau}_2[\mathbf{in}\ \tau_i\ \mathbf{out}\ \tau_o]$
$\top$	$\top$
$\dot{\tau}_1 \cap \dot{\tau}_2$	$\dot{\tau}_1[\mathbf{in}\ \tau_i\ \mathbf{out}\ \tau_o] \cap \dot{\tau}_2[\mathbf{in}\ \tau_i\ \mathbf{out}\ \tau_o]$
$\mathtt{C}\langle \mathbf{in}\ \dot{\tau}_i\ \mathbf{out}\ \dot{\tau}_o\rangle$	$\mathtt{C}\langle \mathbf{in}\ \dot{\tau}_i[\mathbf{in}\ \tau_o\ \mathbf{out}\ \tau_i]\ \mathbf{out}\ \dot{\tau}_o[\mathbf{in}\ \tau_i\ \mathbf{out}\ \tau_o]\rangle$

Figure 3.8: Declarative and Reductive Literal Subtyping Rules for Ceylon (assuming Material-Shape Separation)

argument to the parameter, and the **out** argument being the *covariant* argument (see also Section 2.3). As an example, the literal

$$\mathtt{MutableList}\langle \mathbf{in}\ \mathtt{Integer}\langle \mathbf{in}\ \bot\ \mathbf{out}\ \top\rangle\ \mathbf{out}\ \mathtt{Number}\langle \mathbf{in}\ \bot\ \mathbf{out}\ \top\rangle\rangle$$

represents a mutable list that one can put integers *in*to and get numbers *out* of. The classes `Integer` and `Number` make no use of their type parameter, so in such cases we will use $C\langle\rangle$ as shorthand for $C\langle \mathbf{in}\ \bot\ \mathbf{out}\ \top\rangle$. Thus the example literal will be written as $\mathtt{MutableList}\langle \mathbf{in}\ \mathtt{Integer}\langle\rangle\ \mathbf{out}\ \mathtt{Number}\langle\rangle\rangle$.

Figure 3.8 shows the declarative and reductive literal subtyping rules for Ceylon. They assume a given relation $C\langle\cdot\rangle <:: C'\langle\dot{\tau}\rangle$ indicating that class C

with type parameter $\cdot$ directly inherits class C' with type argument $\dot{\tau}$, where the grammar for parameterized types $\dot{\tau}$ is the same as for types τ with a single additional case $\cdot$ representing the type parameter. Because use-site variance supplies *two* type arguments to each type parameter, substitution $\dot{\tau}[\textbf{in } \tau_i \textbf{ out } \tau_o]$ substitutes all occurrences of $\cdot$ in $\dot{\tau}$ with τ_i in contravariant positions and τ_o in covariant positions. We show the subtyping rules in inference-rule format above for readability, but of course these presentations could be reformatted as collections of literal subtyping rules $\mathcal{D}_{\text{Ceylon}}$ and $\mathcal{R}_{\text{Ceylon}}$.

Similar to how subtyping for functions worked in our toy functional language, we had to merge the two rules for inheritance and variance into a single rule for reductive subtyping. The reductive literal subtyping rule is clearly syntax-directed. Its use of the reflexive-transitive closure $\leq::$ of the inheritance relation $<::$ makes it satisfy the literal reflexivity and transitivity requirements. The declarative and reductive rules admit each other, leaving only the well-foundedness requirement, which we solve by adopting Material-Shape Separation as discussed in Chapter 2. All of this implies that declarative and reductive subtyping for Ceylon are equivalent and decidable by the Decidability of Declarative Subtyping Theorem.

But declarative and reductive subtyping are incapable of the reasoning needed to support the various features of Ceylon discussed in Section 3.2. The remainder of this section illustrates how to extend subtyping with the necessary new functionality. We first discuss each feature independently, then show to use our Intersector Composability Theorem to unify all

these features into one coherent and principled subtyping system. As we proceed, we make use of the following shorthands:

$$\mathsf{ClassOf}(C\langle \textbf{in } \tau_i \textbf{ out } \tau_o\rangle) = C \qquad C\langle\rangle = C\langle \textbf{in } \bot \textbf{ out } \top\rangle$$

$$\frac{C\langle\cdot\rangle <:: C'\langle\dot{\tau}\rangle}{C <:: C'} \qquad \frac{C\langle\cdot\rangle \leq:: C'\langle\dot{\tau}\rangle}{C \leq:: C'}$$

3.6.2 *Disjointness*

Ceylon reasons about when intersections of two types are uninhabitable. We can add disjointness reasoning to a nominal type system using the extension in Figure 3.9. First, we assume we are given a decidable and sound, but not necessarily complete, relation between classes indicating when two classes are disjoint. We further assume this relation is symmetric and respects inheritance. Second, we extend subtyping to say that, whenever two classes are disjoint, any intersection of their instantiations is a subtype of $\bot$. Third, we say that a list of literals is intersected if it does not contain any disjoint classes, and we define our intersector to replace any list of literals containing disjoint classes with $\bot$. Lastly, we prove that this definition satisfies the requirements outlined in Section 3.4. These proofs are straightforward, so we omit them here and throughout this section.

GIVEN a decidable relation C dsj C'

$$\frac{C \text{ dsj } C'}{C' \text{ dsj } C} \qquad \frac{C <:: C' \quad C' \text{ dsj } C''}{C \text{ dsj } C''}$$

EXTENDED SUBTYPING

$$\frac{C \text{ dsj } C'}{C\langle \textbf{in } \tau_i \textbf{ out } \tau_o\rangle \cap C'\langle \textbf{in } \tau_i' \textbf{ out } \tau_o'\rangle <: \bot}$$

INTEGRATED SUBTYPING

$$\phi_{\mathsf{dsj}}(\vec{\ell}) = \nexists \ell, \ell' \in \vec{\ell}.\ \mathsf{ClassOf}(\ell) \text{ dsj } \mathsf{ClassOf}(\ell')$$

$$\bigsqcap_{\mathsf{dsj}} \vec{\ell} = \begin{cases} \bigcap \vec{\ell} & \text{if } \phi_{\mathsf{dsj}}(\vec{\ell}) \\ \bot & \text{otherwise} \end{cases}$$

Figure 3.9: Disjointness Extension

3.6.3 *Principal Instantiation*

Given an intersection $C\langle \textbf{in } \tau_i \textbf{ out } \tau_o\rangle \cap C\langle \textbf{in } \tau_i' \textbf{ out } \tau_o'\rangle$, Ceylon can combine the covariant and contravariant type arguments to generate the *principal instantiation* $C\langle \textbf{in } \tau_i \cup \tau_i' \textbf{ out } \tau_o \cap \tau_o'\rangle$ of class C for that intersection.

Figure 3.10 formalizes this extension. The formalization actually assumes a hierarchy satisfying single-instantiation inheritance, but this is only because our simplification lacks the declaration-site variance necessary to express the more flexible concept of principal-instantiation inheritance [King, 2013]. In either case, the extension to subtyping is the same: contravariant type arguments are combined with unions, and covariant type arguments are combined with intersections. Integrating, on the other hand, is much more complicated, though most of the complexity is just tedium. The intersector needs to scale to arbitrary-sized lists of literals, and more importantly it must deal with the fact that inheritance can make

$$\frac{C\langle\cdot\rangle \leq:: C'\langle\dot{\tau}\rangle \qquad C\langle\cdot\rangle \leq:: C'\langle\dot{\tau}'\rangle}{\dot{\tau} = \dot{\tau}'} \quad \text{Given}$$

Extended Subtyping

$$\overline{C\langle\textbf{in } \tau_i \textbf{ out } \tau_o\rangle \cap C\langle\textbf{in } \tau_i' \textbf{ out } \tau_o'\rangle <: C\langle\textbf{in } \tau_i \cup \tau_i' \textbf{ out } \tau_o \cap \tau_o'\rangle}$$

Integrated Subtyping

$$\mathsf{PMeet}_C(\tau, \tau') = \begin{cases} \top & \text{if } \tau = \tau' = \top \\ \tau & \text{if } \tau' = \top \\ \tau' & \text{if } \tau = \top \\ C\langle\textbf{in } \tau_i \cup \tau_i' \textbf{ out } \tau_o \cap \tau_o'\rangle & \text{if } \bigwedge \begin{array}{r} \tau = C\langle\textbf{in } \tau_i \textbf{ out } \tau_o\rangle \\ \tau' = C\langle\textbf{in } \tau_i' \textbf{ out } \tau_o'\rangle \end{array} \end{cases}$$

$$\mathsf{PInst}_C(\tau) = \begin{cases} \top & \text{if } \tau = \top \\ \mathsf{PMeet}_C(\mathsf{PInst}_C(\ell), \mathsf{PInst}_C(\bigcap \vec{\ell})) & \text{if } \tau = \ell \cap \bigcap \vec{\ell} \\ C\langle\textbf{in } \dot{\tau} \textbf{ out } \dot{\tau}\rangle[\textbf{in } \tau_i \textbf{ out } \tau_o] & \text{if } \bigwedge \begin{array}{r} \tau = C'\langle\textbf{in } \tau_i \textbf{ out } \tau_o\rangle \\ C'\langle\cdot\rangle \leq:: C\langle\dot{\tau}\rangle \end{array} \\ \top & \text{otherwise} \end{cases}$$

$$\mathsf{Cover}(\vec{\ell}) = \bigcup_{\mathcal{C} \subseteq \{\mathsf{ClassOf}(\ell) \mid \ell \in \vec{\ell}\}} \mathsf{LCA}_{<::}(\mathcal{C})$$

(where $\mathsf{LCA}_{<::}$ is least common-ancestors w.r.t. $<::$)

$$\phi_{\mathsf{PInst}}(\vec{\ell}) = \forall C.\ \bigcap \vec{\ell} \leq \mathsf{PInst}_C(\bigcap \vec{\ell})$$
$$\bigsqcap_{\mathsf{PInst}} \vec{\ell} = \bigcap_{C \in \mathsf{Cover}(\vec{\ell})} \mathsf{PInst}_C(\bigcap \vec{\ell})$$

Figure 3.10: Principal-Instantiation Extension

the contributors to a principal instantiation arise indirectly[1]. The Cover function is used to identify the set of classes that can have such an indirect contribution. In Ceylon, Cover needs to be more advanced to address other features of the language, but for our simplified language we can just use least common-ancestors.

We should remark that this feature can also be applied to plain function types with the following:

$$\bigsqcap(\tau_i \to \tau_o, \tau_i' \to \tau_o') = (\tau_i \cup \tau_i') \to (\tau_o \cap \tau_o')$$

Such an extension is actually strictly more expressive than that of Forsythe. However, while this this extension is sound for Ceylon due to Ceylon's nominal nature, it is in fact unsound for Forsythe due to Forsythe's structural nature. Thus it is interesting and important that our framework can express both Ceylon's permissive extension and Forsythe's restrictive extension precisely.

3.6.4 *Classes with Enumerated Cases*

Ceylon provides algebraic data types by allowing classes to explicitly enumerate all permitted subclasses if desired. For example, the `Nat` data type could be defined as follows:

1 For example, if $C_\ell\langle\cdot\rangle <:: C\langle\cdot\rangle$ and $C_r\langle\cdot\rangle <:: C\langle\cdot\rangle$, then the type $C_\ell\langle\textbf{in}\ \tau_i\ \textbf{out}\ \tau_o\rangle \cap C_r\langle\textbf{in}\ \tau_i'\ \textbf{out}\ \tau_o'\rangle$ must be a subtype of C's principal instantiation $C\langle\textbf{in}\ \tau_i \cup \tau_i'\ \textbf{out}\ \tau_o \cap \tau_o'\rangle$ despite not *directly* mentioning class C.

GIVEN an operator $\mathsf{cases}(C)$ that, if **defined**,
specifies a finite and computable set of classes

$$\frac{C' \in \mathsf{cases}(C)}{C' <:: C \quad \wedge \quad \mathsf{cases}(C') \textbf{ undefined}}$$

$$\frac{\mathsf{cases}(C) \textbf{ defined} \qquad C'\langle\cdot\rangle <:: C\langle\dot{\tau}\rangle}{C' \in \mathsf{cases}(C) \quad \wedge \quad \dot{\tau} = \cdot}$$

EXTENDED SUBTYPING

$$\frac{\mathsf{cases}(C) \textbf{ defined}}{C\langle\textbf{in } \tau_i \textbf{ out } \tau_o\rangle <: \bigcup_{C' \in \mathsf{cases}(C)} C'\langle\textbf{in } \tau_i \textbf{ out } \tau_o\rangle}$$

INTEGRATED SUBTYPING

$$\mathsf{expand}(C\langle\textbf{in } \tau_i \textbf{ out } \tau_o\rangle) = \begin{cases} \bigcup_{C' \in \mathsf{cases}(C)} C'\langle\textbf{in } \tau_i \textbf{ out } \tau_o\rangle & \mathsf{cases}(C) \textbf{ defined} \\ C\langle\textbf{in } \tau_i \textbf{ out } \tau_o\rangle & \text{otherwise} \end{cases}$$

$$\phi_{\mathsf{cases}}(\vec{\ell}) = \nexists \ell \in \vec{\ell}.\ \mathsf{cases}(\mathsf{ClassOf}(\ell)) \textbf{ defined}$$

$$\bigsqcap_{\mathsf{cases}} \vec{\ell} = \mathrm{DNF}_{\cap}\left(\bigcap_{\ell \in \vec{\ell}} \mathsf{expand}(\ell)\right)$$

Figure 3.11: Enumerated-Cases Extension (assuming Material-Shape Separation for cases)

```
abstract class Nat of Zero, Succ⟨Nat⟩ { ... }

class Zero extends Nat { ... }

class Succ⟨out N extends Nat⟩ extends Nat { ... }
```

Because an enumeration is provided, Ceylon makes `Nat⟨⟩` a subtype of `Zero⟨⟩ ∪ Succ⟨Nat⟨⟩⟩`. The same could be done for a `LinkedList` class with `Nil` and `Cons` cases. This is particularly useful in the context of `switch` statements because it unifies the reasoning for union types and sum types.

We formalize this extension in Figure 3.11. Here, a class C is a class with enumerated cases if the function $\mathsf{cases}(C)$ is defined. If defined, this func-

tion returns a finite set of classes. Any class in that set must directly inherit from C and not itself be a class with enumerated cases (as a simplification to avoid reasoning about chains of expansions). Conversely, if a class has enumerated cases, then any class that directly inherits from it must be in the set of cases and must not change the type arguments (in order to avoid the complexities of generalized algebraic data types). Again, this formalization is overly restrictive due to the lack of declaration-site variance, bounded type parameters, and limited arity, but it still captures the key components. Note that the intersector needs to apply $\mathrm{DNF}_\cap$ because the expansions of cases could introduce a union inside the intersection. This is fine, though, because one can easily show that $\mathrm{DNF}_\cap$ will not reintroduce unintegrated intersections.

Lastly, we chose the particular example of `Nat` for a reason. Note that in our example $\texttt{Nat}\langle\rangle$ gets expanded to $\texttt{Zero}\langle\rangle \cup \texttt{Succ}\langle\texttt{Nat}\langle\rangle\rangle$, which in turn contains $\texttt{Nat}\langle\rangle$. Although not supported by the simplification presented here, this example illustrates another reason why it is important to integrate $\bigsqcap$ into the recursive algorithm itself, rather than eagerly expand a type recursively, since an eager of expansion of $\texttt{Nat}\langle\rangle$ would continue forever. However, with integration comes the problem that the termination measure might increase. In the case of this extension, we convert class literals to literals of *sub*classes, which may have a higher termination measure, as is the case for the measure that we used in Chapter 2. We address this by assuming Material-Shape Separation *for cases*. By this we mean that all of the declared superclasses (besides C) of the cases of a class C are well-defined class types prior to the declaration of C. This enables the measure of C to incorporate the measures of its cases so that replacing C with the union of its cases does not increase the measure.

Ceylon can furthermore adapt the "swap" measure from Tate, Leung, and Lerner [2011], which intuitively measures how often types can "swap sides" due to contravariance as subtyping recurses. One can easily show that material-shape separation ensures that this "swap" measure is also a well-founded measure for reductive subtyping. The effect of using the "swap" measure instead is that the intersector need only ensure that it does not introduce new forms of contravariance in order to ensure decidability of integrated subtyping. For example, because the class `Succ` is covariant, the type $\texttt{Nat}\langle\rangle$ can be expanded to $\texttt{Zero}\langle\rangle \cup \texttt{Succ}\langle\texttt{Nat}\langle\rangle\rangle$ even though the expanded type again refers to $\texttt{Nat}\langle\rangle$.

3.6.5 *Object and Null*

$$\frac{\textsc{Given}\quad \overline{C \leq:: \texttt{Object} \quad \vee \quad C \leq:: \texttt{Null}}}{\textsc{Extended Subtyping}\quad \overline{\top <: \texttt{Object}\langle\rangle \cup \texttt{Null}\langle\rangle}}$$

$$\textsc{Integrated Subtyping}$$

$$\bigsqcap_{\mathsf{ObjNull}} \vec{\ell} = \begin{cases} \bigcap \vec{\ell} & \text{if } \phi_{\mathsf{ObjNull}}(\vec{\ell}) \\ (\texttt{Object}\langle\rangle \bigcap \vec{\ell}) \cup (\texttt{Null}\langle\rangle \bigcap \vec{\ell}) & \text{otherwise} \end{cases}$$
$$\phi_{\mathsf{ObjNull}}(\vec{\ell}) = \exists \texttt{C}\langle \textbf{in } \tau_i \textbf{ out } \tau_o\rangle \in \vec{\ell}$$

Figure 3.12: Object-Null Extension

In Ceylon, every value belongs to either `Object` or `Null`. Consequently, `Object` and `Null` are essentially the enumerated cases of $\top$. Even though $\top$ is not itself a class, we can still express this using an integrator, as shown in

Figure 3.12. For this, a list of literals is intersected if it contains a class literal. In our simplified calculus, this is equivalent to saying the intersection is nonempty, but in Ceylon literals might also be type variables (which will be discussed in Section 3.7.4). The intersector, then, checks this condition and, if it fails to hold, simply distributes the intersection of literals through the union $\texttt{Object}\langle\rangle \cup \texttt{Null}\langle\rangle$. Thus in particular the empty intersection $\top$ becomes $\texttt{Object}\langle\rangle \cup \texttt{Null}\langle\rangle$. Interestingly, the Ceylon team actually avoided adding this feature because $\top$ is essentially everywhere and it was not clear that the original informal reasoning behind our technique safely applied to such an omnipresent type. But now our formalized reasoning and mechanical verification confidently illustrate that this is in fact the easiest extension to verify. 8

3.6.6 *Composing Features*

Lastly, we compose the extensions together to develop the subtyping system and decision procedure for our simplified Ceylon in Figure 3.13. This composition assumes we have made an explicit separation of classes and interfaces. With this separation in place, we enforce single inheritance of classes, which we then employ to provide a specific disjointness relation for the disjointness extension. Furthermore, because we also assume only classes can have enumerated cases, we can prove that different cases are disjoint from each other. This is used to prove that principal instantiation does not reintroduce classes with enumerated cases into non-disjoint intersections. Consequently, the intersector for each extension can be shown to preserve the fact that intersections are intersected according to all the

GIVEN a decidable predicate $\mathsf{class}(C)$

$$\frac{C <:: C_1 \quad \mathsf{class}(C_1) \qquad C <:: C_2 \quad \mathsf{class}(C_2)}{\mathsf{class}(C) \quad \wedge \quad C_1 = C_2} \qquad \frac{\mathsf{cases}(C)\ \textbf{defined}}{\mathsf{class}(C)} \qquad \frac{C \leq:: \mathtt{Null} \leq:: C'}{C = C'}$$

DEFINITION OF DISJOINTNESS

$$\frac{C \neq \mathtt{Null}}{C\ \mathsf{dsj}\ \mathtt{Null}} \qquad \frac{C \neq \mathtt{Null}}{\mathtt{Null}\ \mathsf{dsj}\ C} \qquad \frac{\mathsf{class}(C) \quad \neg\, C \leq:: C' \qquad \mathsf{class}(C') \quad \neg\, C' \leq:: C}{C\ \mathsf{dsj}\ C'}$$

INTEGRATED SUBTYPING

$$\bigsqcap = \bigsqcap_{\text{Ceylon}} ; \bigsqcap_{\text{cases}} ; \bigsqcap_{\text{dsj}} ; \bigsqcap_{\text{PInst}} ; \bigsqcap_{\text{ObjNull}}$$

$$\text{where } (\textstyle\bigsqcap_1 ; \bigsqcap_2)\,\vec{\ell} = \mathrm{DNF}_{\bigsqcap_2}\left(\bigsqcap_1 \vec{\ell}\right)$$

Figure 3.13: Simplified-Ceylon Subtyping System (using the extensions in Figures 3.9 through 3.12)

previously applied extensions, thereby satisfying Requirement 3.13 (Intersected Preservation). We chose to apply the extensions in the given order so as to ensure this property. Thus we have compositionally built a decidable subtyping system for Ceylon, with most of our proof effort focused specifically on the extensions rather than the previously established features of generics with variance due to our Decidability of Extended Subtyping Theorem.

3.7 VARIATIONS, GENERALIZATIONS, RELATED WORK, AND FUTURE WORK

The technique we have presented is actually a specialization to union and intersection types [Coppo and Dezani-Ciancaglini, 1978; Coppo, Dezani-

Ciancaglini, and Sallé, 1979; Pottinger, 1980; Wijngaarden et al., 1975] of a more general framework we developed that is independent of union and intersection types. We chose to present this particular specialization because it abstracts away much of the complex underlying proof theory, making the technique more accessible while still being applicable to the original language the framework was invented for. Nonetheless it is important to convey some sense of the more general framework, especially as it pertains to existing research and languages.

3.7.1 *Miminal Relevant Logic and Relaxing Requirements*

Bakel, Dezani-Ciancaglini, de'Liguoro, and Motohama [2000] discovered that the propositions-as-types interpretation of **B**+ [Routley and Meyer, 1972], a logic known as minimal relevant logic, corresponds to a call-by-value λ-calculus with subtyping, union, and intersection types. In particular, it corresponds to our toy functional language extended with $\mathcal{E}_{\mathsf{Out}}$ along with the following extension $\mathcal{E}_{\mathsf{In}}$ that was also postulated by Pierce [1991] for Forsythe:

$$\frac{}{(\tau_1 \to \tau') \cap (\tau_2 \to \tau') <: (\tau_1 \cup \tau_2) \to \tau'}$$

Minimal relevant logic is known to be decidable [Gochet, Gribomont, and Rossetto, 2005; Viganò, 2000], which provides a subtyping algorithm for our toy functional language with both $\mathcal{E}_{\mathsf{Out}}$ and $\mathcal{E}_{\mathsf{In}}$ extensions. A slight

variant of our technique can also implement this extension using the following intersector:

$$\prod_{\mathsf{InOut}} \vec{\ell} = \bigcap_{\varnothing \subset \mathcal{L} \subseteq \{L \mid \varnothing \subset L \subseteq \vec{\ell}\}} \left(\bigcap_{L \in \mathcal{L}} \bigcup_{(\tau_i \to \tau_o) \in L} \tau_i\right) \to \left(\bigcup_{L \in \delta(\mathcal{L})} \bigcap_{(\tau_i \to \tau_o) \in L} \tau_o\right)$$

$$\text{where } \delta(\mathcal{L}) = \{\{\sigma(L) \mid L \in \mathcal{L}\} \mid \sigma \text{ is a function from } \mathcal{L} \text{ to } L\}$$

The necessary variations are two parts. First, minimal relevant logic has no $\bot$ type, so we have to remove that from the type system entirely. Second, because of the order in which the lemmas in Section 3.4.5.3 are proven, Requirement 3.12 (Literal Promotion) can be relaxed a bit. In particular, the required proofs with assumptions demonstrating literal promotion can actually be proofs with assumptions *and monotonicity*, meaning they can use the following rule in addition to all the rules in Figure 3.3 (replacing $\mathcal{P} \vdash \tau^{\leftarrow} \leq \tau^{\rightarrow}$ with $\mathcal{P} \vdash_{\leq} \tau^{\leftarrow} \leq \tau^{\rightarrow}$):

$$\frac{\tau_1 \leq \tau_2 \qquad \mathcal{P} \vdash_{\leq} \tau_2 \leq \tau_3 \qquad \tau_3 \leq \tau_4}{\mathcal{P} \vdash_{\leq} \tau_1 \leq \tau_4}$$

With these two variations, one can easily (in a relative sense) show that $\bigcap_{\mathsf{InOut}}$ satisfies the requirements of our framework relative to the $\mathcal{E}_{\mathsf{Out}}$ and $\mathcal{E}_{\mathsf{In}}$ extensions, and as such integrated subtyping $\leq_{\bigcap_{\mathsf{InOut}}}$ provides a decision procedure for extended subtyping $<:_{\mathcal{E}_{\mathsf{Out}} \cup \mathcal{E}_{\mathsf{In}}}$.

Furthermore, essentially the same proof of the requirements applies to the variant of $\bigcap_{\mathsf{InOut}}$ in which $\mathcal{L}$ is allowed to be empty. This variant implements the additional extension $\top <: \top \to \top$, which was postulated by Barbanera, Dezani-Ciancaglini, and de'Liguoro [1995], and which Liquori and Stolze [2017] recently proved decidable. Thus in all of the

above variations on extensions, subtyping has already been proven decidable. However, each proof used a new algorithm and a customized proof technique. Our technique can handle them all in a unified manner, and at the same time the proof is easy to adapt to even further extensions.

3.7.2 *Integrators: Beyond Union and Intersection Types*

On that point, our generalized framework extends even beyond union and intersection types. In place of $\mathrm{DNF}_{\sqcap}$ and $\mathrm{dnf}_{\sqcap}$, one can use an arbitrary type integrator $\int$ and integrated predicate. In place of union and intersection subtyping rules, one can abstractly formulate all the subtyping rules much like we did for literal rules. This generalized framework soundly and completely extends subtyping algorithms with the ability to recognize additional axioms. The technique we presented is a specialization of that framework to distributive union and intersection types. Interestingly, by specializing we were able to relax some of the requirements imposed by our general framework, and we are working on integrating these insights back into the generalized framework.

3.7.3 *Predicative Higher-Rank Polymorphism and Duality*

Our framework can even be dualized, with the *right*-hand type being *co*integrated rather than the left-hand type being integrated, and this dual variant also has prior applications. In predicative higher-rank polymorphism, Peyton Jones, Vytiniotis, Weirich, and Shields [2007] transformed right-hand types into weak-prenex form in order to address

a problem in earlier work by Odersky and Läufer [1996] that did not recognize the following distributivity law described by Mitchell [1988]: $\forall\alpha.\,\tau_1 \to \tau_2 <: \tau_1 \to \forall\alpha.\,\tau_2$ (when α is not free in τ_1). Reynolds' own algorithm for Forsythe (without unions) transforms the types *on both sides* into intersections of "simple types" [Reynolds, 1988, 1997]. But the type transformer satisfies the requirements of a *co*integrator, and so our framework indicates the algorithm could be optimized to only transform the right-hand type.

3.7.4 *Bounded Type Variables and Well-Formed Kind Contexts*

Another aspect of subtyping we have not yet discussed is contexts. For example, languages with subtyping typically also have a kind context of bounded type variables. Ceylon itself is an example of such a language. We have proven that our technique extends to languages with type variables that can be both lower- and upper-bounded, assuming the variable-constraint hierarchy is well-founded in accordance with Material-Shape Separation (Chapter 2).

Although this has been only proven by hand, we present the extension of our framework to bounded type variables here both for completeness with respect to our primary application, Ceylon, and to illustrate an interesting subtlety in this extension. The grammar of types τ is extended with type variables α. Kind contexts Θ are simply lists of type-variable constraints $\tau < \alpha < \tau$. However, in order to ensure decidability, we must restrict these type-variable constraints in accordance with Material-Shape Separation (Chapter 2). That is, the upper- and lower-bounds of a type

Extended Kind Context Validity $\vdash_{\sqcap} \Theta$

$$\frac{}{\vdash_{\mathcal{E}} \cdot} \qquad \frac{\Theta \vdash_{<:\mathcal{E}} \tau_\ell \qquad \Theta \vdash_{<:\mathcal{E}} \tau_u \qquad \vdash_{\mathcal{E}} \Theta \qquad \Theta \vdash \tau_\ell <:_{\mathcal{E}} \tau_u}{\vdash_{\mathcal{E}} \Theta, \tau_l < \alpha < \tau_u}$$

Extended Subtyping with Bounded Type Variables $\Theta \vdash \tau <:_{\mathcal{E}} \tau$

$$\left(\begin{array}{c}\text{All rules in Figure 3.1,} \\ \text{replacing } \tau^{\leftarrow} <: \tau^{\rightarrow} \\ \text{with } \Theta \vdash \tau^{\leftarrow} <: \tau^{\rightarrow}\end{array}\right) \qquad \frac{\tau_\ell < \alpha < \tau_u \in \Theta}{\Theta \vdash \tau_\ell <:_{\mathcal{E}} \alpha} \qquad \frac{\tau_\ell < \alpha < \tau_u \in \Theta}{\Theta \vdash \alpha <:_{\mathcal{E}} \tau_u}$$

Figure 3.14: Extended Subtyping with Bounded Type Variables

variable α must not themselves refer to α. In order to formalize this, we use a type-validity judgement $\Theta \vdash_S \tau$ that is defined in the obvious manner from a literal-validity judgement $\Theta \vdash_S \ell$ specified by the particular type system at hand to indicate when all the free type variables in a literal ℓ are declared in Θ (where S is a parameter representing the subtyping relation).

Extended subtyping with bounded type variables is formalized in Figure 3.14. On the left, we define kind-context validity. There are two key aspects to note here. First, when adding a new constrained type variable α to a kind context Θ, one checks that the bounds are valid under Θ *without* α. This captures Material-Shape Separation as applied to type variables (although we do not assume the literals here represent classes). Second, one checks that the lower bound is *already* a subtype of the upper bound under Θ *without* the new constraint. This ensures that the new constraint does not indirectly introduce any subtypings between types that do not reference α. For example, under the kind context $\top < \alpha < \bot$, transitivity

Integrated Kind Context Validity $\vdash_\sqcap \Theta$

$$\frac{}{\vdash_\sqcap \cdot} \qquad \frac{\Theta \vdash_{\leq_\sqcap} \tau_\ell \qquad \Theta \vdash_{\leq_\sqcap} \tau_u \qquad \vdash_\sqcap \Theta \qquad \Theta \vdash \tau_\ell \leq_\sqcap \tau_u}{\vdash_\sqcap \Theta, \tau_l < \alpha < \tau_u}$$

Integrated Subtyping $\Theta \vdash \tau \leq_\sqcap \tau$
with Bounded Type Variables $\Theta \vdash \tau \leq^\sqcap \tau$

$$\frac{\Theta \vdash \mathrm{DNF}^\Theta_\sqcap(\tau) \leq^\sqcap \tau'}{\Theta \vdash \tau \leq_\sqcap \tau'} \qquad \left(\begin{array}{c}\text{All rules in Figure 3.2 except}\\ \text{literal subtyping, replacing}\\ \tau^\leftarrow \leq \tau^\rightarrow \text{ with } \Theta \vdash \tau^\leftarrow \leq^\sqcap \tau^\rightarrow\end{array}\right)$$

$$\frac{\tau_\ell < \alpha < \tau_u \in \Theta \qquad \Theta \vdash \tau \leq^\sqcap \tau_\ell}{\Theta \vdash \tau \leq^\sqcap \alpha} \qquad \frac{}{\Theta \vdash \alpha \leq^\sqcap \alpha} \qquad \frac{r \in \mathcal{R} \qquad \forall \langle \tau, \tau' \rangle \in \mathcal{P}_r.\ \Theta \vdash \mathrm{DNF}^\Theta_\sqcap(\tau) \leq^\sqcap \tau'}{\Theta \vdash \ell_r^\leftarrow \leq^\sqcap \ell_r^\rightarrow}$$

Figure 3.15: Integrated Subtyping with Bounded Type Variables

would imply that all types are subtypes of each other, even if none of the types involved mention α. On the right, we define extended subtyping with the standard declarative rules for bounded type variables.

Integrated subtyping with bounded type variables is formalized in Figure 3.15. It adds two of the standard reductive subtyping rules for bounded type variables. However, surprisingly, it is missing the rule for upper bounds of type variables. This is because we discovered that upper bounds were actually best achieved by using the integrator rather than a rule. The reason is that the integrator already needs to recurse through the upper bounds of type variables in order to fulfill its responsibilities. For example, while the type $\alpha \cap \texttt{Null}\langle\rangle$ might appear to be already integrated, if α has $\texttt{Object}\langle\rangle$ as its upper bound then the integrator needs to replace the entire type with $\bot$ in order to properly implement disjointness. Thus the definition of DNF^Θ_c, shown in Figure 3.16, recurses into the upper

$$\mathrm{DNF}^{\Theta}_{\sqcap}(\tau) = \mathrm{DNF}^{\Theta}_{\lambda\vec{\ell},\vec{\alpha}.\,\mathrm{DNF}_{\lambda\vec{\ell}'.(\bigcap\vec{\ell}')\cap(\bigcap\vec{\alpha})}(\bigsqcap\vec{\ell})}(\tau)$$

τ	$\mathrm{DNF}^{\Theta}_{c}(\tau)$ where $c : \vec{\ell} \times \vec{\alpha} \to \tau$
$\bot$	$\bot$
$\tau_1 \cup \tau_2$	$\mathrm{DNF}^{\Theta}_{c}(\tau_1) \cup \mathrm{DNF}^{\Theta}_{c}(\tau_2)$
$\top$	$c([], [])$
$\tau_1 \cap \tau_2$	$\mathrm{DNF}^{\Theta}_{\lambda\vec{\ell}_1,\vec{\alpha}_1.\,\mathrm{DNF}^{\Theta}_{\lambda\vec{\ell}_2,\vec{\alpha}_2.c(\vec{\ell}_1 +\!\!+ \vec{\ell}_2,\vec{\alpha}_1 +\!\!+ \vec{\alpha}_2)}(\tau_2)}(\tau_1)$
ℓ	$c([\ell], [])$
α	$\mathrm{DNF}^{\Theta}_{\lambda\vec{\ell},\vec{\alpha}.c(\vec{\ell},[\alpha] +\!\!+ \vec{\alpha})}(\tau_u)$ where $\tau_\ell < \alpha < \tau_u \in \Theta$

Figure 3.16: Definition of $\mathrm{DNF}^{\Theta}_{\sqcap}$

bounds of variables, collecting the list of literals *and* type variables that need to be intersected together. Finally, $\mathrm{DNF}^{\Theta}_{\sqcap}$ takes the collected literals, passes them to the intersector, and then adds the collected type variables to each intersection in the resulting sum of products. The consequence of this is that the upper bound of a type variable is already incorporated into the result of the integrator, making the upper-bound rule for type variables unnecessary.

With a few requirements formulating standard expectations about the literal-validity judgement $\Theta \vdash_S \ell$, one can prove that integrated subtyping with bounded type variables is both decidable and equivalent to extended subtyping with bounded type variables. However, we have yet to mechanically prove these facts. As for the more general framework, we have an abstract formulation of contexts for which kind contexts of bounded type variables are simply a special case.

3.7.5 *Julia and Changing Kind Contexts*

An even greater challenge we have yet to approach, though, is contexts that change as the algorithm recurses. For example, Julia [Bezanson et al., 2017] has a `UnionAll` type constructor that is essentially upper-and-lower-bounded existential quantification [Zappa Nardelli et al., 2018]. As the algorithm recurses, it unpacks these existential types when they are on the left-hand side, adding their type variables and respective bounds to the kind context. To make matters even more difficult, in order to determine how to pack existential types when they are on the right-hand side, the algorithm returns inferred constraints on the to-be-instantiated type variables. If extended to handle these issues, our framework could likely solve an important open problem for Julia. In particular, Julia subtyping attempts to support essentially the following axioms regarding its covariant tuple types:

$$\frac{}{(\tau_1 \cup \tau_1') \times \tau_2 <: (\tau_1 \times \tau_2) \cup (\tau_1' \times \tau_2)}$$

$$\frac{}{(\exists \tau_\ell \preccurlyeq \alpha \preccurlyeq \tau_u.\, \tau_1) \times \tau_2 <: \exists \tau_\ell \preccurlyeq \alpha \preccurlyeq \tau_u.\, (\tau_1 \times \tau_2)}$$

However, largely due to these axioms, it is not yet known whether Julia subtyping is decidable or even transitive. Both those properties are especially important for Julia because Julia uses subtyping checks to resolve multimethod invocations, so decidability would ensure these run-time decisions would behave as expected, and transitivity would enable Julia to optimize its resolution strategies. These axioms can be implemented by integrating the left-hand type to pull unions and existentials out of tuples,

so extending our framework to this setting of changing input and output contexts would enable us to prove for certain that Julia subtyping is both decidable and transitive.

3.7.6 *Regular-Coinductive Subtyping*

Another way changing input contexts are used in subtyping is to track which subtyping goals are currently in progress. This is useful for *regular* subtyping systems [Bonchi and Pous, 2013; Brandt and Henglein, 1997; Gesbert, Genevès, and Layaïda, 2015; Hosoya, Vouillon, and Pierce, 2000; Kozen, Palsberg, and Schwartzbach, 1995], which are particularly common in domain-specific languages [Acay and Pfenning, 2017; Ancona and Corradi, 2016; C. J. Anderson et al., 2014; Henglein and Nielsen, 2011; Jeannin, Kozen, and Silva, 2017] and are also used in C# [Hejlsberg, Wiltamuth, and Golde, 2005; Kennedy and Pierce, 2007; Viroli, 2000]. Whereas well-founded subtyping rules ensure recursive proof search terminates, regular subtyping rules ensure the recursion eventually repeats itself. Thus tracking in-progress goals enables the recursive search to identify infinite loops, which are the only form of infinite recursion in regular systems, and terminate accordingly. Since our proofs for the most part simply rely on the recursion principle of proof search, it seems likely we could extend our framework to this setting. Ancona and Corradi [2016] recently achieved decidability for their subtyping system by transforming the types on both sides, and extending our framework might ensure that they could achieve decidability more efficiently by transforming only the left-hand type.

3.7.7 *Semantic Subtyping*

The work by Ancona and Corradi [2016] mentioned above has another important quality: it is an example of *semantic* subtyping [Frisch, Castagna, and Véronique Benzaken, 2002; Hosoya and Pierce, 2003]. In semantic subtyping one constructs a set-theoretic model and an interpretation of types as sets in that model; two types are then considered to be subtypes whenever their corresponding sets in the model are subsets. The challenge, then, is to show that this subtype relation derived from the model is decidable for the type system at hand, and there has been impressive work achieving this for various type systems [Ancona and Corradi, 2016; Castagna and Xu, 2011; Dardha, Gorla, and Varacca, 2013; Frisch, Castagna, and Véronique Benzaken, 2002, 2008; Hosoya and Pierce, 2003]. In particular, the work on semantic subtyping provides powerful reasoning for union and intersection types due to their obvious correspondence to unions and intersections of sets, and is able to recognize subtypings such as the above example regarding Julia's union and covariant tuple types. In fact, the work on semantic subtyping can even reason about *negative* types [Aiken and Wimmers, 1993; Frisch, Castagna, and Véronique Benzaken, 2002], where the type $\neg\tau$ represents the set of values *not* represented by τ. However, we found that semantic subtyping was ill-suited to our setting for both methodological and technological reasons.

On the methodological side, semantic subtyping is fundamentally tied to a model. But in a setting such as Ceylon, that model is constantly changing. It changes because the exact design of the Ceylon language changes (especially in the early stages). It changes because the exact

design of various Ceylon libraries change, with fields and methods of classes and interfaces regularly being added, removed, or modified as projects evolve. So with semantic subtyping, the set of programs that are accepted would change with each of these changes. This is to be expected—the problem, though, is in *how* this set of programs changes. For example, adding new functionality to the language can cause once-valid programs to become invalid because the new functionality happens to change the model in such a way that types whose interpretations in the model used to coincide no longer do. Similarly, adding a new method to a class can make that class no longer equivalent to another class and consequently programs whose validity unwittingly relied on that equivalence unexpectedly become invalid. And unfortunately one cannot simply limit the reasoning provided by semantic subtyping because the proofs that the derived subtype system is well-behaved (e.g. transitive) are themselves derived from the set-theoretic nature of the model and do not easily adapt to such limitations. Thus we found we needed a declarative system so that we could explicitly state which rules we support now *and commit to supporting in the future*.

On the technological side, we encountered issues with nominality, generics, and decidability. Ceylon is a nominal object-oriented language. It uses nominality as a tool for modularity. For example, one can add a method to a class and, because doing has no impact on the name of this class, know that this will not cause previously-valid programs to become invalid. Unfortunately the work on semantic subtyping has often overlooked or oversimplified the nominal aspects of subtyping. For example, although Dardha, Gorla, and Varacca [2013] provide a nominal semantic model for subtyping, their model assumes a closed world and so will treat in-

terfaces without common implementing classes as disjoint even when there is no (declared) guarantee disallowing developers from adding such common implementing classes in the future. Furthermore their encoding does not extend to reified generics with variance. Putting nominality aside, a more fundamental issue is that generics enable developers to write classes whose corresponding structural types are *not* regular. Regularity has been a fundamental assumption of the semantic-subtyping literature, and generics violate that assumption. In fact, this violation likely ensures that semantic subtyping is undecidable for the resulting model. Thus languages with generics *must* limit which subtypings they recognize. The declarative methodology enables designers to pick which subtypings they believe are most important, and our framework provides a tool for quickly assessing which extensions can be implemented reliably.

3.8 SUMMARY

The framework discussed in this chapter provides us with tools to play around with powerful type-system features built around union and intersection types while keeping decidability. This is not only useful for programming language design in general, it is particularly useful for our approach to gradual typing. As we will see in Chapter 8, joins and meets provided by union and intersection types are one way to get generic type-argument inference more in line with gradual typing. In addition, designing efficient gradually typed languages requires the ability to make modifications to type system features to make them interact well with each other and efficient to implement. In that regard, this framework lets

us re-use much of the work that has already been done before the modification, and gives us a straightforward way to ensure that decidability is not lost in the process.

Part II

IMPLEMENTING GRADUAL TYPING EFFICIENTLY

4

OVERVIEW

As indicated by the title of this dissertation, the main goal is to demonstrate that gradual typing can be implemented efficiently, at least if the language, its type system and its runtime are co-designed with that goal in mind. The basic intuition of how to achieve sound gradual typing is relatively simple: we must protect the guarantees obtained through static type-checking by inserting run-time checks at locations in the code where values flow from untyped components to typed components. If a value from untyped code fails to have its expected type at run time, an exception is thrown. Thus, the statically checked components of the program can assume all values passed to them are well-typed.

Painted this way, the picture leads us to expect that gradual typing may incur some overhead for those inserted checks, proportional to the number of times we transition from untyped to typed parts of the program. In the optimal scenario, these checks are infrequent and efficient, and thus the overall cost of gradual typing is low and can be easily estimated and planned for by analyzing the level of interaction between typed and untyped parts of the program. Furthermore, typed code can be optimized in ways untyped code cannot, so one would expect performance to smoothly improve as types are added to a code base. However, to date, the only existing proposals for a sound gradually typed language that have such performance behavior and reasonable performance for fully untyped code

are the ones discussed in Chapters 5 and 6. The fact that sound gradual typing has an efficiency problem was brought to the forefront by Takikawa et al. [2016], wo measured extreme and unpredictable dips in performance for programs consisting of both typed and untyped code. These measurements were made on their current proposal for sound gradual typing for Racket [Tobin-Hochstadt and Felleisen, 2006], but they argued that no other system provides convincing reasons for why it should perform significantly better.

In this part, we address the problem of achieving efficient sound gradual typing. The core component of this approach is a nominal type system with run-time type information, which lets us verify assumptions about many large data structures with a single, quick check – Chapter 5 demonstrates that in a purely nominal language, called *Nom*, gradual typing is indeed very efficient. The downside of this approach is that it limits expressivity, particularly with respect to structural data. While one should be able to build useful programming languages even under these limitations, we address some of these limitations in Chapter 6, where we add the possibility to have structural data in untyped code and *monotonically* cast it to nominal interfaces in a language called *MonNom*, based on earlier ideas on monotonic references [Rastogi et al., 2015; Siek, Vitousek, Cimini, Tobin-Hochstadt, et al., 2015; Swamy et al., 2014; Vitousek, Kent, et al., 2014]. Furthermore, by designing both systems hand-in-hand with gradual typing, we are able to execute even untyped code efficiently despite our reliance on nominal typing.

To support our claims about efficiency, we built prototype compilers for both languages object-oriented language and used it to implement key benchmarks. As expected, Nom is particularly efficient, measuring

worst-case overheads of less than 10% relative to the performance of untyped code on benchmarks where Takikawa et al. measured overheads of over 10,000%. MonNom's nominal implementation is about on par with Nom, and while its structural values are – as would be expected – slower than their nominal counterparts, they are still reasonable efficient: our key benchmark for structural values sufferes about 30% overhead relative to untyped code.

4.1 BACKGROUND ON GRADUAL TYPING

Gradual typing, as originally proposed by Siek and Taha [2006], features two core elements: a special type **dyn** and a consistency relation $\tau \sim \tau'$ expressing that τ and τ' are structurally equal except for places featuring **dyn**. For example, the following types are consistent:

$$\mathbf{dyn} \sim (\mathbf{dyn} \to \mathbf{dyn}) \sim (\mathbf{int} \to \mathbf{dyn}) \sim (\mathbf{int} \to \mathbf{bool})$$

A program in their gradually typed language type-checks according to the original typing rules, but with type equality replaced with the consistency relation in many places. This enables **dyn** to stand for any type while also maintaining the familiar static typing rules where **dyn** is not present. The gradually typed program is then translated into a variant of the original statically typed language by inserting dynamic casts where run-time checks are necessary to monitor the boundary between untyped and typed code.

Here is how this works for an example program:

$$f : \mathbf{dyn} \to \mathbf{dyn} \vdash (\lambda f' : \mathbf{int} \to \mathbf{int}.\ f'\ 5)\ f$$

The lambda term expects a parameter of type $\mathbf{int} \to \mathbf{int}$, but it is applied to an argument of type $\mathbf{dyn} \to \mathbf{dyn}$. Despite this difference in types, the program type-checks because these two types are considered to be consistent with each other. This gradually typed program is then translated to a statically typed program by inserting run-time casts, resulting in

$$f : \mathbf{dyn} \to \mathbf{dyn} \vdash (\lambda f' : \mathbf{int} \to \mathbf{int}.\ f'\ 5)\ (f :: \mathbf{int} \to \mathbf{int})$$

Here $e :: \tau$ represents a run-time check that e has type τ, conceptually throwing a run-time exception if it does not.

4.1.1 *Casting Strategies*

In most gradual typing settings, casts are the only source of run-time overhead incurred by gradual typing. Thus, where casts are inserted and how they work has a big impact on the performance of a gradually typed program. Before we suggest our own variation on casts in Sections 5.2 and 6.6, we give an overview of existing casting strategies that have been studied as such.

4.1.1.1 *Guarded*

Most work on sound gradual typing—including the original works by Siek and Taha [2006], Tobin-Hochstadt and Felleisen [2006], Matthews and

Findler [2007], and Gronski et al. [2006]—uses the *guarded* cast semantics. In those systems, a cast like the one above reduces as follows:

$$f :: \textbf{int} \to \textbf{int} \quad \mapsto \quad \lambda x : \textbf{int}.\ (f\ (x :: \textbf{dyn})) :: \textbf{int}$$

Instead of checking whether the function f always returns an **int** when given an **int**, which is generally impossible, it is wrapped in a new function that upcasts its input to **dyn**—which always works—and, after the call to f completes, checks that its output is an **int**. Wadler and Findler [2009], and later Ahmed et al. [2011], showed that this is sound even if it is later discovered that f does not always return an **int** when given an **int**. However, instead of having one check at the point where the function is passed to the typed part of the program, this strategy will incur checks every time the function is called, which can cause signficant overhead if that function is heavily used. Simply wrapping functions into other functions also does not preserve object identity, which can be a problem in languages where object identity is semantically significant.

4.1.1.2 *Transient*

The *transient* cast semantics was proposed by Vitousek, Kent, et al. [2014] to preserve object identity in Reticulated Python. It puts casts nearly everywhere in the code: the caller of a function casts an argument to the type that the function expects, but since a different caller might see that function as $\textbf{dyn} \to \tau$, the function itself also casts its parameters, leading to many unnecessary checks even in fully typed code. As such, soundness was not originally meant to be monitored in production programs, but rather intended to help with finding the sources of type errors during

debugging. However, Vitousek, Swords, and Siek [2017] recently used this casting strategy as the basis of their work on open-world soundness, finding overheads much smaller than those reported by Takikawa et al. [2016], but still several multiples of the original run times, sometimes over 10x.

4.1.1.3 *Monotonic*

Another approach used by Vitousek, Kent, et al. [2014] in Reticulated Python, and by Swamy et al. [2014] and Rastogi et al. [2015] in Safe TypeScript, is what Siek, Vitousek, Cimini, Tobin-Hochstadt, et al. [2015] formalized as the *monotonic* approach. Here, every value keeps track of what type it has been checked to have, and enforces that type in later mutations. For example, the record $\{x : 5,\ y : \text{“}Hello''\}$ might be checked to have type $\{x : \mathbf{int}\}$, after which it will get a special run-time-type-information field assigned and become $\{x : 5,\ y : \text{“}Hello'',\ rtti : \{x : \mathbf{int}\}\}$. Any subsequent assignment of a non-integer value to x would fail, and future checks can use the information in *rtti* to fast-track failure or success instead of checking the value of x itself. The *rtti* field itself can only change monotonically towards more precise types (if they are consistent with the current values in the structure). Applying this scheme to higher-order types is not straightforward; thus Swamy et al. do not treat a function $\mathbf{dyn} \to \mathbf{dyn}$ as compatible with $\mathbf{int} \to \mathbf{int}$, while Siek et al. fall back to guarded semantics for function types. In Chapter 6, we use ideas from this line of work to integrate unannotated lambdas and records into our otherwise nominal type system.

4.1.2 *Properties of Gradual Type Systems*

Beyond soundness, there are additional desired properties for gradual type systems suggested by the literature. In the following, we describe what they are and why they are useful. Later, in Section 5.2, we propose two more properties related specifically to the efficiency of gradual typing.

4.1.2.1 *Blame and Accountability*

Lacking a proper word for a notion of "can assign blame correctly" as defined by Tobin-Hochstadt and Felleisen [2006] and Wadler and Findler [2009], we define *accountability* as the property that, when an inserted cast fails, it can refer the programmer to some untyped part of the program that is at fault. Higher-order types are what make blame hard to implement, since a higher-order cast cannot be determined right or wrong until later in the program when the cast function is supplied an argument. *Blame tracking* is the technique used to enable dynically created casts to keep track of the statically inserted cast they originated from.

4.1.2.2 *The Gradual Guarantee*

Siek, Vitousek, Cimini, and Boyland [2015] defined the *gradual guarantee*, which expresses the idea that adding or removing type information from a program should not change its behavior in unexpected ways. In particular, making a well-typed program more dynamic should always result in a well-typed program that produces the same output. The only exception is that a more dynamic program can succeed where the original would fail

because the original might assert some unnecessary and overly restrictive type cast.

The gradual guarantee thus captures the expectation that adding type annotations to an untyped program should preserve the semantics of the program *provided* those annotations are correct. While this clearly seems like a desirable property for gradually typed languages, Siek, Vitousek, Cimini, and Boyland [2015] demonstrate that several existing gradual type systems do not satisfy this property, including Safe TypeScript [Swamy et al., 2014]. They remark that it seems "challenging to satisfy the gradual guarantee and efficiency at the same time".

4.1.3 *Overhead of Gradual Typing*

A few years ago, Takikawa et al. [2016] surveyed the state of performance evaluations on gradual type systems. They found that no gradually typed language had a systematic evaluation of the behavior of the language during the process of gradually typed software development, by which they mean an evaluation of how having mixed typed and untyped code affects run-time overheads. What they found instead was that if there was some kind of overhead evaluation, it usually just compared completely typed and completely untyped versions of programs. Thus, Takikawa et al. proposed a scheme of using microbenchmarks divided up into smaller modules. Each of these modules would exist in two versions, one completely typed, and one completely untyped. Thus, if a program consists of N modules, it would have 2^N potential configurations (i.e. different combinations of typed/untyped versions of the modules). Takikawa et al. created

a suite of microbenchmarks in Typed Racket, and measured the overhead of gradual typing by comparing the running time of each configuration to the running time of the completely untyped configuration. While the completely typed configuration was usually about 30% faster than the completely untyped one, they found some programs had configurations with over 10,000% overhead. Furthermore, for some programs there was no sequence of annotating modules (simulating a gradual evolution from a completely untyped to a completely typed program) where every intermediate configuration had less than 1,000% overhead. This result prompted a flurry of follow-on work, among them Chapter 5 [Bauman et al., 2017; Feltey et al., 2018; Greenman and Felleisen, 2018; Kuhlenschmidt, Almahallawi, and Siek, 2019; Richards, Arteca, and Turcotte, 2017; Vitousek, Swords, and Siek, 2017]. At the time of writing this dissertation, the results in Chapter 5 still provide the most efficient system for run-time casts, at the cost of a far more restrictive type system than the other languages have. In Chapter 6 we relax these restrictions significantly while still erring on the side of relative efficiency.

4.1.4 *Gradual Typing for Object-Oriented Languages*

Gradual typing was extended to object-oriented languages quite early, again by Siek and Taha [2007]. Their approach was based on structural subtyping on records. They used the guarded casting strategy, even delaying checks for the presence of expected fields to whenever that field was actually accessed. This is an example of how design choices in casting strategies are not limited to just functions. Since the language we are formalizing

and have implemented is a nominal object-oriented language, it touches on many aspects from prior work on sound gradual object-oriented languages (both nominal and structural). Such languages include C# [Bierman, Meijer, and Torgersen, 2010], GradualTalk [Allende, Callaú, et al., 2014], Reticulated Python [Vitousek, Kent, et al., 2014], Safe TypeScript [Rastogi et al., 2015; Swamy et al., 2014], and StrongScript [Richards, Nardelli, and Vitek, 2015]. We discuss their relations to our work as we get to the relevant parts of this part.

5

EXPLOITING NOMINALITY FOR EFFICIENCY

This chapter is based on a paper presented at OOPSLA 2017: *Sound Gradual Typing is Nominally Alive and Well* [Muehlboeck and Tate, 2017b]. The implementation and experiments discussed in Section 5.8 are available as an artifact [Muehlboeck and Tate, 2017a].

5.1 INTRODUCTION

This chapter is describes an efficient implementation of gradual typing in a purely nominal, object-oriented language, called Nom, and presents a novel theoretical framework to formalize it and its important properties. The chapter is organized as follows:

- We present new desirable properties of sound gradual type systems that we believe significantly improve their performance (Section 5.2).
- We present a simple gradually typed nominal object-oriented language (Section 5.4 through 5.6) that fulfills the properties traditionally desired of gradual type systems in addition to our own new properties (Section 5.7). We also give a crisp connection between the direct semantics of the language (Section 5.5) and the cast semantics of the language (Section 5.6).

- We provide evidence of our approach's feasibility and efficiency by presenting an implementation of said language and comparing benchmarks between it, Typed Racket [Takikawa et al., 2016], C# [Bierman, Meijer, and Torgersen, 2010], and Reticulated Python [Vitousek, Kent, et al., 2014] (Section 5.8).

5.2 TOWARDS WELL-BEHAVED AND EFFICIENT GRADUAL TYPING

In the light of the previous discussion, we want to devise a sound gradually typed language that is accountable, fulfills the gradual guarantee, and has acceptably low overhead for the checks needed to ensure soundness. Since the overhead of gradual typing comes from the run-time checks it needs to insert, we aim to minimize the number and cost of those checks. The main ingredients of our scheme to achieve this goal are nominality and run-time type information. The idea is that every value will be tagged with its most precise type as run-time type information. This enables what we call *transparency* and *immediate* accountability, the combination of which provides efficiency.

In this section, we give a brief overview of what these ingredients are and how our approach relates to existing work. We formalize transparency and immediate accountability in Section 5.7.

5.2.1 *Transparency*

A transparent casting strategy is one in which a cast is invisible to the runtime system after it is evaluated, unless of course it fails. Thus, guarded

casting is not transparent because a cast can wrap a value with a new value that would otherwise not be present. Transient casting, on the other hand, is transparent because the value is simply passed on after the cast succeeds. Monotonic casting provides a middle ground in which the same value is passed on, but the value is modified in place.

5.2.2 *Immediate Accountability*

Accountability is the ability to identify a source of a cast failure in the source program. *Immediate* accountability is the ability to identify that source immediately as it is being executed. In other words, loops and recursion aside, once execution has successfully proceeded past a point in the program, then that point cannot be at fault for some future cast failure. None of guarded casting, transient casting, or monotonic casting are necessarily immediately accountable. They often only do shallow aspects of a cast immediately, and defer deep aspects of a cast to later. C# [Bierman, Meijer, and Torgersen, 2010] and Safe TypeScript [Swamy et al., 2014] are the only prior gradual type systems that we know of that are immediately accountable, both of which sacrifice the gradual guarantee to achieve this.

5.2.3 *Run-Time Type Information*

Having every value always be tagged with its most precise type requires a significant assumption: every value's most-precise type must be known upon construction of the value, even if it is constructed in an untyped part

of the program. We discuss the implications of this requirement next and in Section 7.3.

5.2.4 *Discussion*

Most earlier work on gradually typing focuses on adding gradual typing to an existing system. Half of this work aims to add gradual typing to an existing untyped language. Examples of this category are Reticulated Python [Vitousek, Kent, et al., 2014], Gradualtalk [Allende, Callaú, et al., 2014], Safe TypeScript [Rastogi et al., 2015; Swamy et al., 2014], and StrongScript [Richards, Nardelli, and Vitek, 2015], as well as the two widespread unsound gradually typed languages (preferably referred to as *optionally* typed languages [Bracha, 2004]), Hack [Facebook, 2016] and TypeScript [Microsoft, 2012]. The other half aims to add gradual typing to, i.e. "gradualize", an existing typed language. Examples of this category are C# [Bierman, Meijer, and Torgersen, 2010], gradual typing for generics by Ina and Igarashi [2011], and work on systematically [Garcia, Clark, and Tanter, 2016] or automatically [Cimini and Siek, 2016] gradualizing given typed languages.

Certainly much of the appeal of gradual typing is that it can give a pre-existing language new access to the counterpointing paradigm. However, both directions currently have weaknesses to overcome due to the fact that gradual typing is heavily intertwined with both the type system and the runtime implementation. Adding sound gradual typing to an untyped language seems to frequently incur significant overhead, sometimes making programs multiple orders of magnitude slower [Takikawa et al., 2016].

Part of the problem is that the type-system features needed to capture the idioms common to untyped languages are not easy to check efficiently, especially when the underlying runtime is not designed for it.

Conversely, adding gradual typing to a typed language can introduce unexpected behavior due to violations of the gradual guarantee. For example, in C#, adding more precise type information to a well-typed program may cause that program to cease being well-typed, as the new information may introduce ambiguities (e.g. through additional available overloadings) that would have to be resolved. When such an ambiguity is introduced at compile time, C# can rely on the programmer to resolve the error. However, with gradual typing, such ambiguities can be introduced at run time, where no such programmer is readily available to resolve the problem, causing the system to throw a run-time error. Furthermore, C# compilation is heavily type-directed, but gradual typing often makes type information available only at run time, so C# is forced to defer much of its compilation of untyped code to run time. We have found that this can introduce significant overhead, as we illustrate in Section 5.8. We discuss these and other issues in more detail in Chapter 7. The main point here is that gradual typing is not easy to bolt onto existing languages without serious drawbacks.

Thus, in contrast to most earlier work, we focus on gradual typing for new systems, where the entire language can be designed from the start to both support and benefit from gradual typing. Clearly we can benefit from all the work on adding gradual typing to existing systems, but our change in focus also enables us to benefit from a greater degree of flexibility. Here we use that flexibility to address the efficiency issues in prior work while retaining desirable properties such as accountability and the gradual guar-

antee. While the improvement in performance is certainly more noticeable when compared to systems that have added sound gradual typing to untyped languages, we even achieve better performance than systems that have added sound gradual typing to typed languages. We accomplish this by designing a language with a nominal runtime environment, which is where most of our performance gains come from, optimized for gradual typing, which is where our smaller performance gains come from. Nominality in and of itself is not a guarantee for good performance, nor does it imply transparency or accountability. For example, our benchmarks for C#—which is nominal, transparent, and immediately accountable—show that its dynamically typed parts are quite slow (see Section 5.8). As another example, StrongScript [Richards, Nardelli, and Vitek, 2015] uses nominality for performance in fully typed programs, but the language as a whole is neither transparent nor immediately accountable, and there is no performance evaluation of mixed programs, where Takikawa et al. [2016] found the biggest problems. Furthermore, Richards et al. found that blame tracking produced significant overhead, prompting them to only evaluate the performance of their system without blame.

Of course, the nominality of our runtime environment restricts the programmer. While gradual typing can recover some of the expressiveness of structural typing that prior research has worked hard to preserve, there is still much that is lost. We expect to address this by developing methods for mixing structural values into our nominal system, much like we mix untyped and typed code—Chapter 6 discusses first steps in this direction based on the work here. In fact, there is already significant work to this effect, some with [Richards, Nardelli, and Vitek, 2015; Wrigstad et al., 2010] and some without gradual typing [C. Anderson and Drossopoulou, 2003].

But it is important to recognize that adding structural reasoning is not necessary for many of the well-known applications of gradual typing. As we alluded to in Chapter 0, one envisions gradual typing being a part of the software-development process from the beginning. Stable code would typically be typed, benefiting from better optimization and providing machine-checkable document for programmers and IDEs interacting with this code. Meanwhile unstable code would not need to be typed, which is useful for prototyping, scripting, or simply letting the programmer first experiment in the paradigm they are most comfortable with. In particular, student programmers can enjoy the benefits of working with well-typed APIs without having the type system impede their first explorations into programming.

What we present in this chapter is a minimal system striving towards this end, just large enough to test whether this path has promise. Our formalization presented in Section 5.4 is sufficient for covering the same feature set as Featherweight Java [Igarashi, Pierce, and Wadler, 2001] with interfaces and little more. Meanwhile, we have made an effort to be forwards compatible with a multitude of features frequently found in nominal industry languages, all while also making an effort to be forwards compatible with structural values. Our implementation covers a much larger subset of pre-generics Java, including assignment, interfaces, overloading, primitive types, messages to super, access control, and null pointers. Some of these features were adapted to work with gradual typing in a way that satisfies the gradual guarantee. For example, we require that all overloadings of a method be disjoint in order to avoid ambiguities at dynamic method lookup at run time, and we made null explicit in anticipation of adding generics types later to avoid problems

with both type-argument inference [Smith and Cartwright, 2008] and unsoundness [Amin and Tate, 2016].

5.3 THE OPTIMISTIC PERSPECTIVE

Throughout the remainder of this chapter we will be using the terms *optimistic* and *pessimistic*. This is a change in terminology that we find unifies our definitions. The idea is that there are two attitudes towards typing. One is the optimistic attitude: programs should be able to proceed so long as they might succeed. Dynamic typing takes this attitude, trying to only stop a program when execution encounters an issue that cannot be overcome. The other is the pessimistic attitude: programs should only be able to proceed when it is known they will succeed. Static typing takes this attitude, trying to only compile a program if it is exhibits certain guarantees.

Both attitudes have their advantages and disadvantages, and consequently are each better suited to different circumstances. The purpose of gradual typing is to give the programmer the ability to explicitly control which attitude is applied where in a given program. Thus, when a variable is given the type **`dyn`**, the programmer is directing the compiler to treat the variable as optimistically having whatever type is necessary for the usages at hand. On the other hand, when a variable is given the type **Number**, the programmer is directing the compiler to treat the variable as pessimistically only being usable where a **Number** provides sufficient guarantees. In this way, a gradually typed language enables the programmer to change attitudes as they see fit.

$$
\begin{array}{rr}
\text{Class/Interface Name } C & \text{Field Name } f \\
\text{Method Name } m & \text{Variable Name } x
\end{array}
$$

$$
\begin{array}{rcl}
\text{Type } \tau & ::= & \top \mid C \mid \textbf{dyn} \\
\text{Context } \Gamma & ::= & \cdot \mid \Gamma, \tau\ x \\
\text{Expression } e & ::= & x \mid \textbf{let}\ \tau\ x := e\ \textbf{in}\ e \mid C(e, ...) \mid e.f_\delta \\
& & \mid\ e.m_\delta(e, ...) \mid \textbf{cast}\ e\ \textbf{to}\ \tau \\
\text{Dispatch Mode } \delta & ::= & C \mid \textbf{dyn} \\
\text{Method Signature } s & ::= & \tau\ m(\Gamma) \\
\text{Method Definition } d & ::= & s \mapsto e \\
\text{Interface Definition } i & ::= & \textbf{interface}\ C\ \{s; \dots\} \\
\text{Class Definition } c & ::= & \textbf{class}\ C(\tau\ f, \dots)\ \textbf{implements}\ C, \dots \{d; \dots\} \\
\text{Environment Definition } \Psi & ::= & \cdot \mid \Psi, i \mid \Psi, c
\end{array}
$$

Figure 5.1: Grammar

In technical terms, this new terminology expresses the same concept that Garcia, Clark, and Tanter [2016] generalize as *consistent lifting*.

5.4 THE TYPE SYSTEM

The grammar of our small gradually typed object-oriented language is shown in Figure 5.1. The grammar is mostly standard besides being fairly minimal. Our implementation does of course handle a much richer set of features as described in Section 5.8. The point of this formalization is not to specify our implemented language, but to be able to discuss the interesting aspects of our approach: nominality, run-time type information, transparency, and immediate accountability. Note that as another simplification, we do not concern ourselves with naming issues; it is obvious how to adjust the rules throughout this chapter to address problems such as name shadowing.

5.4.1 *Dispatch Modes*

The one irregular feature of our grammar is the use of *dispatch modes* δ. Every field access and method invocation is annotated with a dispatch mode. This reflects the fact that, at compile time, one must decide how a field should be accessed or a method implementation should be looked up. For example, a method could be looked up by accessing some offset of the object's virtual-method table. In this case, the dispatch mode is the class that specifies which offset to use. Alternatively, a method could be looked up by searching through the object's interface table, in which case the dispatch mode is the interface to search for. Lastly, since we are providing a gradually typed language, a method could be looked up in the object's hashtable, like one would do in a dynamically-typed language such as Python. In this case, the dispatch mode is **`dyn`**.

Note that this means we view objects as supplying both a virtual-method table[1] (and interface table) *and* a (possibly immutable and shared) hashtable. Similarly, fields can be accessed through fixed offsets when the object's class is known, or through the hashtable when the field access is being typed dynamically. This allows us to interact with objects efficiently regardless of the typing attitude we happen to be applying in a given part of the program. Although in theory we could develop a more traditional calculus without dispatch modes, we include them here to better illustrate how we are able to implement gradual typing.

In general, the dispatch modes will be inferred by the compiler. As this issue is orthogonal to the properties that we are trying to formalize

1 For simplicity, we do not allow classes to extend other classes. However, we have designed our calculus to support class inheritance, and our implementation supports it as well.

Subtyping $\Psi \vdash \tau \; S \; \tau$

$$\frac{}{\Psi \vdash C \; S \; C} \qquad \frac{C \; \textbf{implements} \; C' \in \Psi}{\Psi \vdash C \; S \; C'} \qquad \frac{}{\Psi \vdash \tau \; S \; \top}$$

$$\frac{}{\Psi \vdash \tau \; S \; \textbf{dyn}} \qquad \frac{}{\Psi \vdash \textbf{dyn} \triangleleft C}$$

Figure 5.2: Subtyping, where S is optimistic $\triangleleft$ or pessimistic $\blacktriangleleft$

and comes with its own interesting design choices, we defer discussion of dispatch-mode inference to Appendix B.1.

5.4.2 *Subtyping*

Our system provides two kinds of subtyping: optimistic and pessimistic. Optimistic subtyping ($\triangleleft$) recognizes that **dyn** is optimistically a subtype of any type τ because it can optimistically be interpreted as being τ. Pessimistic subtyping ($\blacktriangleleft$) ensures that one type is a subtype of another only if all values of the former type are also values of the latter type. The two differ by only rule, so we use the metavariable S to formalize both of them simultaneously in Figure 5.2.

Like most subtyping relations, pessimistic subtyping is transitive. However, optimistic subtyping, like its inspiration *consistent* subtyping [Siek and Taha, 2007], is *not* transitive because it conceptually confuses existentials with universals. That is, **dyn** semantically represents $\exists\alpha.\alpha$. Consequently, every type is semantically a subtype of **dyn**, as is captured by both pessimistic and optimistic subtyping. But the optimistic attitude says to also treat **dyn** as $\forall\alpha.\alpha$ *when it would make the subtyping hold*, making **dyn** an optimistic subtype of every type. Thus the difference between optimistic

Shorthands $\Psi \vdash_S e \qquad \Psi \vdash e\ S\ \tau$

$$\frac{\Psi \vdash \tau \qquad \Psi \vdash e\ S\ \tau}{\Psi \vdash_S e} \qquad \frac{\Psi \mid \cdot \vdash e\ S\ \tau}{\Psi \vdash e\ S\ \tau}$$

Expression Typing $\Psi \mid \Gamma \vdash e\ S\ \tau$

$$\frac{\tau\ x \in \Gamma \qquad \Psi \vdash \tau\ S\ \tau'}{\Psi \mid \Gamma \vdash x\ S\ \tau'} \qquad \frac{\Psi \mid \Gamma \vdash e\ S\ \top \qquad \Psi \vdash \tau\ S\ \tau'}{\Psi \mid \Gamma \vdash \mathbf{cast}\ e\ \mathbf{to}\ \tau\ S\ \tau'}$$

$$\frac{\Psi \vdash \tau \qquad \Psi \mid \Gamma \vdash e\ S\ \tau \qquad \Psi \mid \Gamma, \tau\ x \vdash e'\ S\ \tau'}{\Psi \mid \Gamma \vdash \mathbf{let}\ \tau\ x := e\ \mathbf{in}\ e'\ S\ \tau'} \qquad \frac{\mathbf{class}\ C(\tau_1, ..., \tau_n) \in \Psi \qquad \forall i.\ \Psi \mid \Gamma \vdash e_i\ S\ \tau_i \qquad \Psi \vdash C\ S\ \tau}{\Psi \mid \Gamma \vdash C(e_1, \ldots, e_n)\ S\ \tau}$$

$$\frac{\Psi \vdash \delta.f : \tau \qquad \Psi \vdash e\ S\ \delta \qquad \Psi \vdash \tau\ S\ \tau'}{\Psi \mid \Gamma \vdash e.f_\delta\ S\ \tau'} \qquad \frac{\Psi \vdash \delta.m(\tau_1, \ldots, \tau_n) : \tau \qquad \Psi \mid \Gamma \vdash e\ S\ \delta \qquad \forall i.\ \Psi \mid \Gamma \vdash e_i\ S\ \tau_i \qquad \Psi \vdash \tau\ S\ \tau'}{\Psi \mid \Gamma \vdash e.m_\delta(e_1, \ldots, e_n)\ S\ \tau'}$$

Field Lookup $\Psi \vdash \tau.f : \tau$

$$\frac{\mathbf{class}\ C(\tau_1\ f_1, \ldots) \in \Psi}{\Psi \vdash C.f_i : \tau_i} \qquad \frac{}{\Psi \vdash \mathbf{dyn}.f : \mathbf{dyn}}$$

Method Lookup $\Psi \vdash \tau.m(\tau, ...) \vdash \tau$

$$\frac{\tau\ C.m(\tau_1, \ldots, \tau_n) \in \Psi}{\Psi \vdash C.m(\tau_1, \ldots, \tau_n) : \tau} \qquad \frac{}{\Psi \vdash \mathbf{dyn}.m(\mathbf{dyn}, \ldots) : \mathbf{dyn}}$$

Type Validity $\Psi \vdash \tau$

$$\frac{}{\Psi \vdash \top} \qquad \frac{\mathbf{interface}\ C \in \Psi}{\Psi \vdash C} \qquad \frac{\mathbf{class}\ C \in \Psi}{\Psi \vdash C} \qquad \frac{}{\Psi \vdash \mathbf{dyn}}$$

Context Validity $\Psi \vdash \Gamma$

$$\frac{}{\Psi \vdash \cdot} \qquad \frac{\Psi \vdash \Gamma \qquad \Psi \vdash \tau}{\Psi \vdash \Gamma, \tau\ x}$$

Figure 5.3: Expression Typing, where S is either optimistic $\triangleleft$ or pessimistic $\blacktriangleleft$ subtyping

and pessimistic subtyping captures the difference between the optimistic and pessimistic attitudes.

5.4.3 *Expression Typing*

Our expression-typing rules are shown in Figure 5.3. Observe that they look nearly identical to what one might expect for a statically typed language. The only other major difference is that they are parameterized by a subtyping relation S. When one uses optimistic subtyping $\triangleleft$ for S, we say the expression type-checks optimistically. Likewise, when one uses pessimistic subtyping $\blacktriangleleft$ for S, we say the expression type-checks pessimistically. This parameterization illustrates that type-checking is both standard and adjustable to the preferred attitude at hand.

5.4.4 *Class and Interface Validation*

Class and interface validation is shown in Figure 5.4. Once again it is quite standard. The one point to note is that a class is allowed to only optimistically satisfy method signatures of implemented interfaces. In this way the class implementation can be completely untyped, even if it is implementing typed interfaces. The only requirement then is that the class specify the list of interfaces it intends to implement, and at least provide methods with the appropriate names and arities. Note also that method definitions are always type-checked optimistically. Consequently, one of the challenges is to achieve sound gradual typing throughout the class hierarchy.

Environment Validity $\vdash \Psi \qquad \Psi \vdash \Psi$

$$\frac{\Psi \vdash \Psi}{\vdash \Psi} \qquad \frac{}{\Psi \vdash \cdot} \qquad \frac{\Psi \vdash \Psi' \qquad \Psi \vdash i}{\Psi \vdash \Psi', i} \qquad \frac{\Psi \vdash \Psi' \qquad \Psi \vdash c}{\Psi \vdash \Psi', c}$$

Interface Validity $\Psi \vdash i$

$$\frac{\forall i.\ \Psi \vdash s_i}{\Psi \vdash \textbf{interface}\ C\ \{s_1; \dots\}}$$

Class Validity $\Psi \vdash c$

$$\frac{\begin{array}{c}\forall i.\ \Psi \vdash \tau_i \qquad \forall i.\ \Psi \mid C \vdash d_i \\ \forall i.\ \textbf{interface}\ C_i\ \{s_1^i; \dots\} \in \Psi \qquad \forall i. \forall j. \exists k_{i,j}.\ \Psi \vdash d_{k_{i,j}} \triangleleft s_j^i\end{array}}{\Psi \vdash \textbf{class}\ C(\tau_1\ f_1, \dots)\ \textbf{implements}\ C_1, \dots\ \{d_1; \dots\}}$$

Signature Validity $\Psi \vdash s$

$$\frac{\Psi \vdash \tau \qquad \Psi \vdash \Gamma}{\Psi \vdash \tau\ m(\Gamma)}$$

Method Definition Typing $\Psi \vdash d$

$$\frac{\Psi \vdash \tau \qquad \Psi \vdash \Gamma \qquad \Psi \mid C\ \textbf{this}, \Gamma \vdash e \triangleleft \tau}{\Psi \mid C \vdash \tau\ m(\Gamma) \mapsto e}$$

Overriding $\Psi \vdash \Gamma \triangleleft \Gamma \qquad \Psi \vdash d \triangleleft s$

$$\frac{}{\Psi \vdash \cdot \triangleleft \cdot} \qquad \frac{\Psi \vdash \Gamma \triangleleft \Gamma' \qquad \Psi \vdash \tau \triangleleft \tau'}{\Psi \vdash \Gamma, \tau\ x \triangleleft \Gamma, \tau'\ x} \qquad \frac{\Psi \vdash \Gamma' \triangleleft \Gamma \qquad \Psi \vdash \tau \triangleleft \tau'}{\Psi \vdash \tau\ m(\Gamma) \mapsto e \triangleleft \tau'\ m(\Gamma')}$$

Figure 5.4: Class and Interface Validation

Value	$v ::= C(v, \dots)$
Error	$\varepsilon ::= v.f_{\mathbf{dyn}} \mid v.m_{\mathbf{dyn}}(v, \dots) \mid \mathbf{cast}\ v\ \mathbf{to}\ C$
Valuation	$\nu ::= v \mid \varepsilon \mid \infty$
Evaluation Context	$E ::= \cdot \mid \mathbf{let}\ \tau\ x := E\ \mathbf{in}\ e$
	$\mid C(v, \dots, E, e, \dots) \mid E.f_\delta \mid E.m_\delta(e, \dots)$
	$\mid v.m_\delta(v, \dots, E, e, \dots) \mid \mathbf{cast}\ E\ \mathbf{to}\ \tau$
Method Implementation	$\bar{d} ::= \tau\ m_\delta(\Gamma) \mapsto e$
Class Implementation	$\bar{c} ::= \mathbf{class}\ C(\tau\ f, \dots)$
	$\mathbf{implements}\ C, \dots\ \{\bar{d}; \dots\}$
Environment Implementation	$\bar{\Psi} ::= \cdot \mid \bar{\Psi}, i \mid \bar{\Psi}, \bar{c}$

Terminals

$$\Psi \vdash e\ \mathbf{terminal}\ \tau \qquad \Psi \vdash e\ \mathbf{erroneous}$$
$$\Psi \vdash e\ \mathbf{bad\text{-}cast} \qquad \Psi \vdash e\ \mathbf{lapse}\ \tau$$

$$\frac{\Psi \vdash v \blacktriangleleft \tau}{\Psi \vdash v\ \mathbf{terminal}\ \tau} \qquad \frac{\Psi \vdash e\ \mathbf{erroneous}}{\Psi \vdash e\ \mathbf{terminal}\ \tau} \qquad \frac{\Psi \vdash e\ \mathbf{bad\text{-}cast}}{\Psi \vdash e\ \mathbf{erroneous}}$$

$$\frac{\Psi \vdash_\blacktriangleleft E \quad \Psi \vdash_\blacktriangleleft v \quad v = C(\dots) \quad \neg\, \Psi \vdash C \blacktriangleleft C'}{\Psi \vdash E[\mathbf{cast}\ v\ \mathbf{to}\ C']\ \mathbf{bad\text{-}cast}} \qquad \frac{\Psi \vdash_\blacktriangleleft E \quad \Psi \vdash_\blacktriangleleft v \quad v = C(\dots) \quad \mathbf{class}\ C(f^1, \dots) \in \Psi \quad \nexists i.\ f = f^i}{\Psi \vdash E[v.f_{\mathbf{dyn}}]\ \mathbf{erroneous}}$$

$$\frac{\nexists e'.\ \Psi \vdash e \twoheadrightarrow e' \quad \neg\, \Psi \vdash e\ \mathbf{terminal}\ \tau}{\Psi \vdash e\ \mathbf{lapse}\ \tau} \qquad \frac{\Psi \vdash_\blacktriangleleft E \quad \Psi \vdash_\blacktriangleleft v \quad \forall i.\ \Psi \vdash_\blacktriangleleft v_i \quad v = C(\dots) \quad \nexists \tau, \tau_1, \dots, \tau_n.\ \tau\ C.m(\tau_1, \dots, \tau_n) \in \Psi}{\Psi \vdash E[v.m_{\mathbf{dyn}}(v_1, \dots, v_n)]\ \mathbf{erroneous}}$$

Figure 5.5: Grammar and Terminal Classification

Valuations $\bar{\Psi} \vdash e\ R^{\infty}\ v : \tau \qquad \Psi \vdash e\ R^{*}\ \mathbf{lapse}\ \tau$

$$\frac{\bar{\Psi} \vdash e\ R^{*}\ v \qquad \bar{\Psi} \vdash v \blacktriangleleft \tau}{\bar{\Psi} \vdash e\ R^{\infty}\ v : \tau} \qquad \frac{\bar{\Psi} \vdash e\ R^{*}\ E[\varepsilon] \quad \bar{\Psi} \vdash E[\varepsilon]\ \mathbf{erroneous}}{\bar{\Psi} \vdash e\ R^{\infty}\ \varepsilon : \tau}$$

$$\frac{\bar{\Psi} \vdash e\ R^{\infty}}{\bar{\Psi} \vdash e\ R^{\infty}\ \infty : \tau} \qquad \frac{\Psi \vdash e\ R^{*}\ e' \qquad \Psi \vdash e'\ \mathbf{lapse}\ \tau}{\Psi \vdash e\ R^{*}\ \mathbf{lapse}\ \tau}$$

Evaluation Context Validity $\Psi \vdash_S E$

$$\frac{}{\Psi \vdash_S \cdot} \qquad \frac{\Psi \vdash_S E}{\Psi \vdash_S E.f_\delta} \qquad \frac{\Psi \vdash_S E}{\Psi \vdash_S E.m_\delta(e_1, \ldots)}$$

$$\frac{\Psi \vdash_S E}{\Psi \vdash_S \mathbf{let}\ \tau\ x := E\ \mathbf{in}\ e} \qquad \frac{\mathbf{class}\ C(\tau_1, \ldots, \tau_n) \in \Psi \quad \forall j.\ \Psi \vdash v_j\ S\ \tau_j \qquad \Psi \vdash_S E}{\Psi \vdash_S C(v_1, \ldots, v_i, E, e_{i+2}, \ldots, e_n)}$$

$$\frac{\Psi \vdash \delta.m(\tau_1, \ldots, \tau_n) : \tau \qquad \Psi \vdash v\ S\ \delta \quad \forall j.\ \Psi \vdash v_j\ S\ \tau_j \qquad \Psi \vdash_S E}{\Psi \vdash_S v.m_\delta(v_1, \ldots, v_i, E, e_{i+2}, \ldots, e_n)} \qquad \frac{\Psi \vdash_S E}{\Psi \vdash_S \mathbf{cast}\ E\ \mathbf{to}\ \tau}$$

Reductions $\bar{\Psi} \vdash E\ R\ E \qquad \bar{\Psi} \vdash e\ R\ e$

$$\frac{\bar{\Psi} \vdash e\ R\ e' \qquad (\bar{\Psi} \vdash_{\triangleleft} E)}{\bar{\Psi} \vdash E[e]\ R\ E[e']} \qquad \frac{(\bar{\Psi} \vdash v \triangleleft \tau)}{\bar{\Psi} \vdash \mathbf{let}\ \tau\ x := v\ \mathbf{in}\ e\ R\ e[x \mapsto v]}$$

$$\frac{v = C(\ldots) \quad C.m_\delta(\tau_1\ x_1, \ldots, \tau_n\ x_n) \mapsto e \in \bar{\Psi} \qquad (\bar{\Psi} \vdash v \triangleleft \delta) \qquad (\forall i.\ \bar{\Psi} \vdash v_i \triangleleft \tau_i)}{\bar{\Psi} \vdash v.m_\delta(v_1, \ldots, v_n)\ R\ e[\mathbf{this} \mapsto v, x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]}$$

$$\frac{v = C(v_1, \ldots) \qquad \mathbf{class}\ C(f^1, \ldots) \in \bar{\Psi} \quad (\bar{\Psi} \vdash v \triangleleft \delta)}{\bar{\Psi} \vdash v.f^i_\delta\ R\ v_i} \qquad \frac{v = C(\ldots) \quad \bar{\Psi} \vdash C \blacktriangleleft \tau \qquad (\bar{\Psi} \vdash v \triangleleft \top)}{\bar{\Psi} \vdash \mathbf{cast}\ v\ \mathbf{to}\ \tau\ R\ v}$$

Figure 5.6: Operational Semantics, where R is either optimistic $\twoheadrightarrow$ (ignoring parenthesized assumptions) or pessimistic $\rightarrow$ (asserting parenthesized assumptions) reduction

5.5 THE DIRECT SEMANTICS

Traditionally, sound gradually typed calculi are formalized using a type-directed translation to a cast calculus [Cimini and Siek, 2016; Henglein, 1994; Siek and Taha, 2006, 2007]. We will do so as well in the next section, but here we first develop an operational semantics directly on our calculus. The intent is to provide an intuitive semantics that programmers can use to reason about how their gradually typed programs will behave without needing to understand the details of when and where casts are inserted and how they are implemented. In the next section, we will demonstrate that there is a strong relationship between these direct semantics and the ones derived from cast insertions.

We formalize the direct semantics of our calculus using rewrite rules, as presented in Figure 5.6, with the grammar and terminal classification in Figure 5.5. This formalization is odd in that some of the assumptions of the various rules are parenthesized. This is because the rules are parameterized by a reduction relation R that can stand for either *optimistic reduction* ($\rightarrowtriangle$) or *pessimistic reduction* ($\rightarrow$). For optimistic reduction, one ignores the parenthesized assumptions, optimistically hoping that the expected invariants of the system hold. For pessimistic reduction, one includes the parenthesized assumptions, pessimistically asserting the expected invariants of the system throughout execution. Obviously pessimistic reduction provides more guarantees, but optimistic reduction is much more efficient. Thus we can gain much from understanding the relationship between these two semantics.

VALUES, VALUATIONS, AND LAPSES The values in our system are instances of classes. The arguments to the class constructor indicate the object's values for the class's fields.

Note that **`error`** is not an expression in our formalization. Instead, we simply let failing casts get stuck. This means even non-value programs can get stuck for both acceptable and unacceptable reasons. For example, a program could be stuck because it is a failed cast, which is acceptable and would be caught by the runtime system. However, a program could also be stuck because it is trying to access a field at a memory offset not provided by the object, which is unacceptable and corresponds to a potentially dangerous memory-access violation. We use the judgement $\Psi \vdash e$ **`terminal`** τ, defined in Figure 5.5, to indicate when e is stuck for an acceptable reason with type τ. In particular, e could be a value of type τ, a failed dynamic field lookup, a failed dynamic method lookup, or a failed cast. As a convenience, we also use the counterpoint judgement $\Psi \vdash e$ **`lapse`** τ to indicate when e is *pessimistically* stuck for a reason unacceptable for type τ, which we call a lapse because it indicates a current violation of some intended invariant.

Each of these cases represents a different observable result of executing a program. We use *valuations* ν to represent the acceptable results. The idea is that, ignoring situations where a program lapses, a program's semantics are the valuations it can result in. Since a program might fail to terminate, we include ∞ as a valuation representing when programs execute forever. We capture valuations with the judgement $\bar{\Psi} \vdash e\ R^{\infty}\ \nu : \tau$, defined in Figure 5.6. As a convenience, we also use the counterpoint judgement $\Psi \vdash e\ R^{*}$ **`lapse`** τ to indicate that e results in some unacceptable lapse rather than an acceptable valuation.

REDUCTIONS Now we discuss the reduction rules in more detail. As we mentioned before, these rules specify both optimistic reduction ($\rightarrow\!\triangleright$), which ignores the parenthesized assumptions, and pessimistic reduction ($\rightarrow\!\blacktriangleright$), which asserts the parenthesized assumptions. Pessimistic reduction of evaluation contexts uses the judgement $\Psi \vdash_{\triangleleft} E$ to ensure that evaluation of expressions only moves on from left to right when the already computed values actually have their expected types. The use of assertions aside, the reduction rules are standard except for one oddity in our semantics for method invocations.

In particular, the assumption $C.m_{\delta}(\tau_1\ x_1, \ldots, \tau_n\ x_n) \mapsto e \in \bar{\Psi}$ looks up class C's *implementation* for method m *and dispatch mode* δ in the environment *implementation* $\bar{\Psi}$. The most important detail of this assumption is the inclusion of the dispatch mode δ in this lookup. This allows class C to provide a different implementation of m for each appropriate dispatch mode. This will enable C to address the fact that its method definition only *optimistically* satisfies the signatures of the interfaces it implements. To understand how, let us consider implementations in more detail.

IMPLEMENTATIONS Whereas our typing rules are defined in the context of an environment *definition*, our reduction rules are defined in the context of an environment *implementation*. The two differ in that the former specifies class definitions, whereas the latter specifies class implementations. A class definition provides a method definition for each method m of the class; a class implementation provides a method implementation for each method m of the class *and each suitable dispatch mode* δ for m. The body of each such method implementation is a slightly adjusted version of the

body of the method definition to account for the corresponding dispatch mode, as we will describe below.

We formalize implementations of definitions in Figure 5.7. The judgement $\Psi \vdash_S \bar{\Psi}$ indicates that $\bar{\Psi}$ is a valid implementation of the environment definition Ψ. Furthermore, if the parameter S is optimistic subtyping ($\triangleleft$), then the body of every method implementation in $\bar{\Psi}$ is optimistically typed. Likewise, if the parameter S is pessimistic subtyping ($\blacktriangleleft$), then the body of every method implementation in $\bar{\Psi}$ is pessimistically typed.

A class implementation $\bar{c}$ is valid for a class definition c if every method implementation in $\bar{c}$ corresponds to some method definition in c *and* every method definition in c has a corresponding method implementation in $\bar{c}$ for each necessary dispatch mode. In particular, there must be an implementation for the dispatch modes corresponding to the class itself and to **dyn** dispatch. Furthermore, if a method definition is used to satisfy some method signature in an interface implemented by the class, then there must be an implementation for the dispatch mode corresponding to that interface. Thus, a class implementation simply specifies the contents of the virtual-method table, interface table, and dispatch hashtable, but with each way to dispatch a given method having its own implementation (employing low-level tricks to keep the size of the executable down).

Each of these method implementations corresponds to the same method definition, and while that implies they are closely related, it does not imply they are identical. First, the signature of a method implementation coincides with the signature corresponding to its own dispatch mode, *not* to its method definition. Second, the body of the method implementation needs to be adjusted to conform with the corresponding signature. For example, consider a method implementation whose dispatch mode is an

Program Implementation Validation $\Psi \vdash_S \bar{\Psi} \qquad \Psi \mid \Psi \vdash_S \bar{\Psi}$

$$\frac{\Psi \mid \Psi \vdash_S \bar{\Psi}}{\Psi \vdash_S \bar{\Psi}} \qquad \frac{}{\Psi \mid \cdot \vdash_S \cdot} \qquad \frac{\Psi \mid \Psi' \vdash_S \bar{\Psi}}{\Psi \mid \Psi', i \vdash_S \bar{\Psi}, i} \qquad \frac{\Psi \mid \Psi' \vdash_S \bar{\Psi} \qquad \Psi \mid c \vdash_S \bar{c}}{\Psi \mid \Psi', c \vdash_S \bar{\Psi}, \bar{c}}$$

Class Implementation Validation $\Psi \mid c \vdash_S \bar{c}$

$$\frac{\begin{array}{c} c = \mathbf{class}\ C(\tau_1\ f_1, \dots)\ \mathbf{implements}\ C_1, \dots\ \{d_1; \dots\} \\ \bar{c} = \mathbf{class}\ C(\tau_1\ f_1, \dots)\ \mathbf{implements}\ C_1, \dots\ \{\bar{d_1}; \dots\} \\ \forall i. \exists j_i.\ \Psi \mid C \mid d_{j_i} \vdash_S \bar{d_i} \qquad \forall i. \exists j_i.\ \Psi \vdash \bar{d}_{j_i} :_C d_i \qquad \forall i. \exists j_i.\ \Psi \vdash \bar{d}_{j_i} :_{\mathbf{dyn}} d_i \\ \forall i.\ \mathbf{interface}\ C_i\ \{s_1^i; \dots\} \in \Psi \qquad \forall i. \forall j. \exists k_{i,j}.\ \Psi \vdash \bar{d}_{k_{i,j}} :_{C_i} s_j^i \end{array}}{\Psi \mid c \vdash_S \bar{c}}$$

Method Implementation Validation $\Psi \mid C \mid d \vdash_S \bar{d}$

$$\frac{\begin{array}{c} d = \tau\ m(\tau_1\ x_1, \dots, \tau_n\ x_n) \mapsto e \\ \bar{d} = \tau'\ m_\delta(\tau_1'\ x_1, \dots, \tau_n'\ x_n) \mapsto \mathbf{let}\ \tau'\ x := e'\ \mathbf{in}\ x \\ \forall i.\ \Psi \vdash \tau_i' \triangleleft \tau_i \qquad \Psi \vdash \tau \triangleleft \tau' \qquad \Psi \mid C\ \mathbf{this}, \tau_1'\ x_1, \dots, \tau_n'\ x_n \vdash e'\ S\ \tau' \\ \Psi \vdash \mathbf{let}\ \tau_1\ x_1 := x_1\ \mathbf{in}\ \dots \mathbf{let}\ \tau_n\ x_n := x_n\ \mathbf{in}\ \mathbf{let}\ \tau\ x := e\ \mathbf{in}\ x \preceq e' : \tau' \end{array}}{\Psi \mid C \mid d \vdash_S \bar{d}}$$

Dispatch Mode Validation $\Psi \vdash \bar{d} :_\delta d \qquad \Psi \vdash \bar{d} :_\delta s$

$$\frac{\Psi \vdash \bar{d} :_\delta s}{\Psi \vdash \bar{d} :_\delta s \mapsto e} \qquad \frac{}{\Psi \vdash \tau\ m_C(\Gamma) \mapsto e :_C \tau\ m(\Gamma)}$$

$$\frac{}{\Psi \vdash \mathbf{dyn}\ m_{\mathbf{dyn}}(\mathbf{dyn}\ x_1, \dots, \mathbf{dyn}\ x_n) \mapsto e :_{\mathbf{dyn}} \tau\ m(\tau_1\ x_1, \dots, \tau_n\ x_n)}$$

Figure 5.7: Implementation Validation, where S is either optimistic $\triangleleft$ or pessimistic $\blacktriangleleft$ subtyping

interface implemented by the class. The body of the method definition is defined in terms of the class's signature for the method, but that signature only *optimistically* satisfies the signature of the method required by the interface.

We address this difference by inserting variable assignments to retype the method parameters and return value according to the method signature. Next, the *refinement* relation ($\preceq$) specifies that the actual method body e' of the implementation is a refinement of the original body wrapped in these retyping expressions, which means the implementation can have casts inserted to check optimistic assumptions made in the method definition. Refinement is a relation, not a procedure, which means the refined expression may have no additional casts at all, or just the right amount to type-check pessimistically (in addition to optimistically), or many more than necessary. We defer detailed discussion of refinement until the next section.

Given an environment definition Ψ, there exists a naïve implementation of Ψ. In particular, because refinement is reflexive, one can simply define every method implementation to be the body of the corresponding method definition modulo retyping the inputs and output. As an abuse of notation, we refer to this naïve implementation as Ψ. If Ψ is a valid environment definition, then it is trivial to prove that Ψ is also a valid *optimistically*-typed implementation of itself.

Similarly, given an environment implementation $\bar{\Psi}$, there often exists a corresponding definition for $\bar{\Psi}$. In particular, one derives a class C's definition of a method from that method's implementation for the dispatch mode C. As an abuse of notation, we refer to this corresponding definition as $\bar{\Psi}$. If $\bar{\Psi}$ is a valid implementation of some valid environment

definition Ψ, then the definition $\bar{\Psi}$ has exactly the same typing information as Ψ.

SOUNDNESS Even without inserting casts or restricting to specific implementations, we can make interesting observations about the behavior of our direct semantics, as proven in Appendix B.2. The first is that typed expressions are guaranteed to be either terminal or reducible:

Theorem 5.5.1 (Progress). *For every environment Ψ and implementation $\bar{\Psi}$ where $\vdash \Psi$ and $\Psi \vdash_S \bar{\Psi}$ hold,*

$$\forall e, \tau. \quad \Psi \vdash e \; S \; \tau \quad \Longrightarrow \quad \Psi \vdash e \; \texttt{terminal} \; \tau \quad \mathit{xor} \quad \exists e'. \quad \bar{\Psi} \vdash e \; R \; e'$$

where S is either optimistic $\triangleleft$ or pessimistic $\blacktriangleleft$ subtyping, and R is either optimistic $\multimap$ or pessimistic $\rightarrow$ reduction.

Note that this theorem states that even an *optimistically* typed expression is either terminal or *pessimistically* reducible. That is, we can even guarantee pessimistic progress for optimistic expressions. Also, note that in order to be `terminal`, every relevant value in v must be *pessimistically* typed. This is ensurable even for optimistically typed expressions because every optimistically typed *value* is necessarily also pessimistically typed.

The second observation we can make is that *pessimistic* typing is preserved by reduction:

Theorem 5.5.2 (Pessimistic-Type Preservation). *For every environment Ψ and implementation $\bar{\Psi}$ where $\vdash \Psi$ and $\Psi \vdash_{\blacktriangleleft} \bar{\Psi}$ hold,*

$$\forall \tau, e, e'. \quad \Psi \vdash \tau \quad \wedge \quad \Psi \vdash e \blacktriangleleft \tau \quad \wedge \quad \bar{\Psi} \vdash e \; R \; e' \quad \Longrightarrow \quad \Psi \vdash e' \blacktriangleleft \tau$$

where R is either optimistic $\twoheadrightarrow_\triangleright$ *or pessimistic* $\twoheadrightarrow_\blacktriangleright$ *reduction.*

Importantly, this states that even optimistic reduction preserves pessimistic typing, which is arguably the whole purpose of pessimistic typing. However, neither form of reduction preserves optimistic typing. Clearly optimistic reduction does not preserve optimistic typing, otherwise we would not be referring to it as *optimistic* typing. But it is surprising that even pessimistic reduction fails to preserve optimistic typing despite the many run-time assertions it makes. To see why, optimistically type the program **`let dyn`** $x \coloneqq$ `"Hello"` **`in`** x `% 10`, and then try to optimistically type the reduction of that program, `"Hello" % 10`. This failure of pessimistic reduction is critical, as it illustrates why inserting casts is necessary to ensure soundness.

The third and final observation we make is that optimistic and pessimistic reduction *coincide* for *pessimistically* typed programs:

Theorem 5.5.3 (Pessimistic Identification)**.** *For every environment* Ψ *and implementation* $\bar{\Psi}$ *where* $\vdash \Psi$ *and* $\Psi \vdash_\blacktriangleleft \bar{\Psi}$ *hold,*

$$\forall e, \tau.\quad \Psi \vdash e \blacktriangleleft \tau \quad\Longrightarrow\quad \forall v.\quad \bar{\Psi} \vdash e \twoheadrightarrow_\triangleright^\infty v : \tau \quad\Longleftrightarrow\quad \bar{\Psi} \vdash e \twoheadrightarrow_\blacktriangleright^\infty v : \tau$$

This means that, for pessimistically typed programs, we can use the more efficient optimistic reduction and yet still enjoy the stronger guarantees of pessimistic reduction. In particular, a pessimistically typed program will never become unacceptably stuck by either semantics, so its observable results are completely described by its set of valuations, which is identical across the two forms of reduction. Again, this is not true for optimistically typed programs. Thus, given an optimistically typed program, we would like a way to interpret it using a "better"-behaved pessimistically typed

program. This is the purpose of cast insertion, or program refinement, which we discuss next.

5.6 THE CAST SEMANTICS

We define the cast semantics for our gradual calculus using *program refinement*. Program refinement is a generalization of cast insertion, the process traditionally used to enforce soundness for gradual type systems [Findler and Felleisen, 2002; Siek and Taha, 2006; Tobin-Hochstadt and Felleisen, 2006]. Whereas cast insertion traditionally specifies how to transform a program by inserting casts, program refinement simply states that two programs are similar but with one having some casts inserted, akin to the similarity relation defined by Tobin-Hochstadt and Felleisen [2006]. That is, refinement specifies no strategy about how to insert casts. A refinement might have too few casts to achieve a particular goal, or more casts than are strictly necessary. This laxity actually makes it easier to reason about refinement, especially with respect to reduction, and in a more uniform manner, especially with respect to typing.

PROGRAM REFINEMENT Program refinement is formalized using the judgement $\Psi \vdash e \preceq \tilde{e} : \tau$, which indicates that the expression $\tilde{e}$[2] is a refinement of e when the expected output type is τ. The formalization of refinement has only one interesting rule, presented below; the other

2 Note that, whereas the grammar for a $\bar{\Psi}$ is different than that for a Ψ, the notation $\tilde{e}$ is not introducing a new grammar. It is simply a convention we employ to help the reader keep track of which expressions are "original" expressions versus "refined" expressions.

Translation Validation $\vdash \Psi \mid e \rightsquigarrow \bar{\Psi} \mid \tilde{e} : \tau$

$$\frac{\vdash \Psi \qquad \Psi \vdash \tau \qquad \Psi \vdash e \triangleleft \tau \qquad \Psi \vdash_{\blacktriangleleft} \bar{\Psi} \qquad \Psi \vdash e \preceq \tilde{e} : \tau \qquad \Psi \vdash \tilde{e} \blacktriangleleft \tau}{\vdash \Psi \mid e \rightsquigarrow \bar{\Psi} \mid \tilde{e} : \tau}$$

Cast Semantics $\vdash \Psi \mid e \rightsquigarrow \bar{\Psi} \mid \tilde{e} : \tau$

$$\frac{\vdash \Psi \mid e \rightsquigarrow \bar{\Psi} \mid \tilde{e} : \tau \qquad \bar{\Psi} \vdash \tilde{e} \twoheadrightarrow^{\infty} v : \tau}{\Psi \vdash e \rightsquigarrow^{\infty} v : \tau}$$

Figure 5.8: Cast Semantics

rules in Appendix B.3 simply allow this rule to be applied throughout the program.

$$\frac{\Psi \vdash e \preceq \tilde{e} : \tau}{\Psi \vdash e \preceq \mathbf{cast}\ \tilde{e}\ \mathbf{to}\ \tau : \tau}$$

This rule is the only rule that lets refinement insert a cast. It states that we can refine a program by inserting a cast to the expected return type τ of the program. By restricting inserted casts to be of precisely this form, we ensure that they only check optimistic assumptions of the original program. In particular, we avoid inserting casts that would introduce run-time errors that have no relationship to the optimism of the original program, say by arbitrarily inserting casts of string expressions to integers.

PROGRAM TRANSLATION We mentioned that refinement is reflexive, but the primary purpose of refinement is translation of optimistically typed programs into pessimistically typed programs. Although refinement does not specify how precisely to implement such a translation, we can combine it with the concepts we have already developed to formalize the concept of a translation. Given an environment definition Ψ and implementation $\bar{\Psi}$,

expressions e and $\tilde{e}$, and type τ, we say we have a well-formed translation if $\vdash \Psi \mid e \rightsquigarrow \bar{\Psi} \mid \tilde{e} : \tau$ holds, as defined in Figure 5.8. That is, a translation is well-formed if the original program $\Psi \mid e$ *optimistically* has type τ, the translated program $\bar{\Psi} \mid \tilde{e}$ is a *refinement* of the original program with expected return type τ, and the translated program *pessimistically* has type τ.

This indicates when we have *a* well-formed translation, but for a given Ψ and e there may be multiple such translations. To this end, we have the following property, proven in Appendix B.3, that all well-formed translations are semantically equivalent (recalling that pessimistically typed programs cannot get stuck in an unacceptable manner):

Theorem 5.6.1 (Translation Irrelevance)**.** *For every* Ψ, $\bar{\Psi}_1$, $\bar{\Psi}_2$, e, $\tilde{e}_1$, $\tilde{e}_2$, *and* τ,

$$\begin{pmatrix} \vdash \Psi \mid e \rightsquigarrow \bar{\Psi}_1 \mid \tilde{e}_1 : \tau \\ \vdash \Psi \mid e \rightsquigarrow \bar{\Psi}_2 \mid \tilde{e}_2 : \tau \end{pmatrix} \implies \forall v.\ \bar{\Psi}_1 \vdash \tilde{e}_1\ R^{\infty}\ v : \tau \iff \bar{\Psi}_2 \vdash \tilde{e}_2\ R^{\infty}\ v : \tau$$

where R is either optimistic $\multimap$ *or pessimistic* $\rightarrow$ *reduction.*

This means that, in order to use well-formed translation as a basis for our cast semantics, we just need some well-formed translation for our given optimistically typed program. Which one we happen to choose is irrelevant. Fortunately, we have the following:

Theorem 5.6.2 (Translation Existence)**.** *For every environment* Ψ, *expression* e, *and type* τ,

$$\vdash \Psi \quad \wedge \quad \Psi \vdash \tau \ \wedge \quad \Psi \vdash e \triangleleft \tau \quad \implies \quad \exists \bar{\Psi}, \tilde{e}. \quad \vdash \Psi \mid e \rightsquigarrow \bar{\Psi} \mid \tilde{e} : \tau$$

Thus every optimistically typed program has a well-formed translation. Defining such a translation is straightforward and tedious, so we defer formal construction to Appendix B.3.

Given that we have both translation irrelevance and existence, we can define the cast semantics for our gradually typed language using the judgement $\Psi \vdash e \rightsquigarrow^{\infty} v : \tau$ defined in Figure 5.8.

SEMANTIC PRESERVATION Now that we know that we can always refine an optimistically typed program into a pessimistically typed program, we want to know that this translation respects the *direct* semantics of the original program in a reasonable manner. We demonstrate this with two observations, the proofs of which can be found in Appendix B.3.

Theorem 5.6.3 (Pessimistic-Valuation Preservation)**.** *For every environment* Ψ*, expression* e*, and type* τ *where* $\vdash \Psi$ *and* $\Psi \vdash \tau$ *and* $\Psi \vdash e \triangleleft \tau$ *hold,*

$$\forall v. \quad \Psi \vdash e \twoheadrightarrow^{\infty} v : \tau \quad \Longrightarrow \quad \Psi \vdash e \rightsquigarrow^{\infty} v : \tau$$

This states that if the direct semantics of our original program can pessimistically produce some result, then translation also produces that result. That is, translation preserves pessimistic valuations. Note that translation does not preserve optimistic valuations, though. This is because a program can happen to optimistically reduce to some value even if it requires repeatedly violating expected invariants of the system throughout the process, and a typical sound gradual type system has no principled way of safely arriving at that haphazard but fortuitous result.

This leads us to wonder what happens when the original program goes awry. In particular, due to pessimistic progress and preservation, we know

that the translation must result in some valuation even if the original program does not. The following gives us some insight into what the valuation must be.

Theorem 5.6.4 (Optimistic-Valuation Reflection)**.** *For every environment* Ψ*, expression* e*, and type* τ *where* $\vdash \Psi$ *and* $\Psi \vdash \tau$ *and* $\Psi \vdash e \triangleleft \tau$ *hold,*

$$\forall v.\quad \Psi \vdash e \rightsquigarrow^{\infty} v : \tau \quad \Longrightarrow \quad \begin{array}{c} \Psi \vdash e \mathrel{-\!\!\triangleright}^{\infty} v : \tau \\ \textit{or} \\ \Psi \vdash v\ \texttt{bad-cast} \quad \wedge \quad \Psi \vdash e \mathrel{-\!\!\triangleright}^{*} \texttt{lapse}\ \tau \end{array}$$

This states that any valuation resulting from translation must also result optimistically from the original program *unless* the valuation is a bad cast catching the fact that the original program would become pessimistically stuck in an unacceptable manner, which Theorem 5.5.1 guarantees can only happen if the original program would become ill-typed. In combination with pessimistic-valuation preservation, this informs us that the cast semantics is essentially the same as the direct semantics *except* that it results in bad casts rather than lapsing.

Thus, with the combination of valuation preservation and reflection, we see that there is a very close relationship between our cast semantics and our direct semantics, one that is common among sound gradual type systems. This suggests that programmers can rely on the more intuitive direct semantics as a reasonable approximation of what the cast semantics provides. There is still some gap, though, since the cast semantics preserves *pessimistic* valuations but reflects *optimistic* valuations. In most gradual type systems, bridging this gap requires understanding the details of where casts are inserted and how they are implemented. In our system, though,

we can actually close that gap. The stronger guarantees in the next section ensure that our cast semantics even reflects pessimistic valuations, showing that programmers need only understand direct pessimistic reduction to anticipate the behavior of our cast semantics.

5.7 THE GUARANTEES

The challenge at hand is to design a gradually typed language that is both principled and efficient. Here we address the principles by formalizing the guarantees that our calculus provides, the proofs of which can be found in Appendix B.4. Afterwards, we will address efficiency by comparing with other similarly principled gradual type systems.

5.7.1 *Immediacy*

Sound gradual typing guarantees that a cast will fail before the program would get stuck in an unacceptable manner. However, most sound gradually typed languages only have this property with respect to optimistic reduction. Our system has a stronger property, which we call *immediacy*, formalized as follows:

Theorem 5.7.1 (Immediacy)**.** *For every* Ψ, $\bar{\Psi}$, e, $\tilde{e}$, *and* τ *where* $\vdash \Psi \mid e \rightsquigarrow \bar{\Psi} \mid \tilde{e} : \tau$ *holds,*

$$\forall e'.\ \begin{pmatrix} \Psi \vdash e \twoheadrightarrow^* e' \\ \Psi \vdash e' \ \texttt{lapse}\ \tau \end{pmatrix} \implies \exists \tilde{e}'.\ \begin{pmatrix} \bar{\Psi} \vdash \tilde{e} \twoheadrightarrow^* \tilde{e}' \\ \bar{\Psi} \vdash \tilde{e}' \ \texttt{bad-cast} \end{pmatrix} \wedge\ \Psi \vdash e' \preceq \tilde{e}' : \tau$$

In the statement of this theorem, we distinguish the clause $\Psi \vdash e' \preceq \tilde{e}' : \tau$. Without this clause, the theorem simply states that the cast semantics results in a bad cast *whenever* the original program would eventually get pessimistically stuck in an unacceptable manner. This is sufficient to strengthen optimistic-valuation reflection into pessimistic-valuation reflection, as we discussed in the previous section. And with the distinguished clause, the theorem furthermore guarantees that the bad cast occurs *immediately* when the original program would get pessimistically stuck.

This is in contrast with most work on sound gradual typing. To see why, consider the following traditional gradually typed program:

$$\texttt{let dyn} \to \texttt{dyn}\ f \coloneqq (\lambda s : \texttt{str}.\ s.\mathsf{length})\ \texttt{in}$$
$$\texttt{let int} \to \texttt{int}\ g \coloneqq f\ \texttt{in}\ \mathit{slow}();\ g\ 5$$

This program can pessimistically reduce in a single step to the following:

$$\texttt{let int} \to \texttt{int}\ g \coloneqq (\lambda s : \texttt{str}.\ s.\mathsf{length})\ \texttt{in}\ \mathit{slow}();\ g\ 5$$

This reduced program, however, can no longer reduce pessimistically. The value $\lambda s : \texttt{str}.\ s.\mathsf{length}$ fails to have the expected type $\texttt{int} \to \texttt{int}$ of the variable g, even optimistically. This clearly indicates a violation of the intended invariants of the program. For a gradual type system to provide immediacy, the cast semantics for this program would have to raise an error at this point in the execution. However, most prior work cannot recognize the error until the call to $g\ 5$ eventually executes.

Interestingly, threesomes [Siek and Wadler, 2010] do raise an error immediately for this example, provided one uses a variant that is what

Siek, Garcia, and Taha [2009] describe as the *eager* error-detection strategy. Furthermore, it has been proven that eager threesomes can be viewed as a cast-insertion implementation of the semantics prescribed by Garcia, Clark, and Tanter [2016] when applied to a gradually typed lambda calculus [Toro and Tanter, 2017]. So it might generally be the case that the semantics prescribed by Garcia, Clark, and Tanter [2016] will always provide immediacy.

5.7.2 *Immediate Accountability*

Accountability is the ability to indicate what component of the program is to blame for a given cast failure observed by the cast semantics of a program, and to furthermore ensure that only dynamically typed components are ever blamed. Like in previous work on blame [Ahmed et al., 2011; Tobin-Hochstadt and Felleisen, 2006; Wadler and Findler, 2009], we can augment our calculus with labels and errors so that, when such a cast failure occurs, it provides a label specifying some optimistic assumption that turned out not to hold at run time. However, we forgo such an augmentation here because, for our calculus, the process is particularly uninteresting.

The reason is that our system is transparent—unlike in most existing accountable systems, casts are not introduced by our operational semantics. This means that casts are only introduced by program refinement and so directly correspond to locations in the original program. All erroneous casts in our semantics have the property that they are casts to a class or interface type, never to **`dyn`**. Program refinement only introduces casts of

an expression to its expected return type, which means the receiver of such a cast must be statically typed. Furthermore, the expression being refined optimistically has that expected return type. If that expression were also statically typed, that would imply the expression also has that expected return type pessimistically. Type preservation would then ensure that this cast would succeed. So the cast can only fail if the expression is dynamically typed. Thus, all erroneous casts not in the original program are necessarily casts from dynamically typed code to statically typed code that were directly inserted by program refinement, making blame trivial to achieve.

But whereas accountability is the property that a failing cast correctly identifies a faulty optimistic assumption in the source code, what we call *immediate* accountability furthermore demands that execution is currently at that point in the source code. That is, optimistic checks either fail immediately or never. This property makes blame tracking completely unnecessary, since immediate accountability guarantees that a cast fails only if that cast itself is to blame. For our system, the reasoning above, in combination with immediacy, ensures that our system provides immediate accountability.

However, in general the combination of immediacy and accountability is not sufficient to provide immediate accountability. This is evidenced by the fact that eager threesomes [Toro and Tanter, 2017] require blame tracking in order to provide accountability [Siek and Wadler, 2010] even though they provide immediacy.

5.7.3 *The Gradual Guarantee*

The gradual guarantee [Siek, Vitousek, Cimini, and Boyland, 2015], in our terms, states that adding optimism to a program should increase the likelihood that the program will type-check and evaluate successfully, and nothing more. We formalize this using an *optimism* relation ($\sqsubseteq$), shown in Figure 5.9, which indicates when two components only differ in terms of degree of optimism, with the right component being the more optimistic of the two. This is traditionally known as a precision relation [Garcia, Clark, and Tanter, 2016; Siek, Vitousek, Cimini, and Boyland, 2015] or naïve subtyping [Wadler and Findler, 2009]. We use the new terminology both to be consistent to with the rest of the chapter and to address the fact that the precision relation is backwards, as noted by its inventors [Siek, Vitousek, Cimini, and Boyland, 2015], since it places the more precise component on what the name suggests should be the less precise side.

The gradual guarantee formally consists of three theorems adapted from [Siek, Vitousek, Cimini, and Boyland, 2015]. Our first theorem states that a program that is already optimistically typed will still be optimistically typed if it is made more optimistic:

Theorem 5.7.2 (Gradual Optimism)**.**

$$\forall \begin{pmatrix} \Psi, \Psi' \\ \Gamma,\ \Gamma' \\ \tau,\ \tau' \\ e,\ e' \end{pmatrix} . \begin{pmatrix} \vdash \Psi \\ \Psi \quad \vdash \Gamma \\ \Psi \quad \vdash \tau \\ \Psi \mid \Gamma \vdash e \triangleleft \tau \end{pmatrix} \wedge \begin{pmatrix} \Psi \sqsubseteq \Psi' \\ \Gamma \sqsubseteq \Gamma' \\ \tau \sqsubseteq \tau' \\ e \sqsubseteq e' \end{pmatrix} \implies \begin{pmatrix} \vdash \Psi' \\ \Psi' \quad \vdash \Gamma' \\ \Psi' \quad \vdash \tau' \\ \Psi' \mid \Gamma' \vdash e' \triangleleft \tau' \end{pmatrix}$$

Type and Context Optimism $\tau \sqsubseteq \tau \qquad \Gamma \sqsubseteq \Gamma$

$$\frac{}{\top \sqsubseteq \top} \qquad \frac{}{C \sqsubseteq C} \qquad \frac{}{\tau \sqsubseteq \mathbf{dyn}} \qquad \frac{}{\cdot \sqsubseteq \cdot} \qquad \frac{\Gamma \sqsubseteq \Gamma' \qquad \tau \sqsubseteq \tau'}{\Gamma, \tau\ x \sqsubseteq \Gamma', \tau'\ x}$$

Valuation and Expression Optimism $\nu \sqsubseteq \nu \qquad e \sqsubseteq e$

$$\frac{}{\infty \sqsubseteq \infty} \qquad \frac{}{x \sqsubseteq x} \qquad \frac{\tau \sqsubseteq \tau' \qquad e_1 \sqsubseteq e_1' \qquad e_2 \sqsubseteq e_2'}{\mathbf{let}\ \tau\ x := e_1\ \mathbf{in}\ e_2 \sqsubseteq \mathbf{let}\ \tau'\ x := e_1'\ \mathbf{in}\ e_2'}$$

$$\frac{\forall i.\ e_i \sqsubseteq e_i'}{C(e_1, \ldots, e_n) \sqsubseteq C(e_1', \ldots, e_n')} \qquad \frac{e \sqsubseteq e' \qquad \delta \sqsubseteq \delta'}{e.f_\delta \sqsubseteq e'.f_{\delta'}}$$

$$\frac{e \sqsubseteq e' \qquad \delta \sqsubseteq \delta' \qquad \forall i.\ e_i \sqsubseteq e_i'}{e.m_\delta(e_1, \ldots, e_n) \sqsubseteq e'.m_{\delta'}(e_1', \ldots, e_n')} \qquad \frac{e \sqsubseteq e' \qquad \tau \sqsubseteq \tau'}{\mathbf{cast}\ e\ \mathbf{to}\ \tau \sqsubseteq \mathbf{cast}\ e'\ \mathbf{to}\ \tau'}$$

Program Optimism $\Psi \sqsubseteq \Psi \qquad i \sqsubseteq i \qquad c \sqsubseteq c$

$$\frac{}{\cdot \sqsubseteq \cdot} \qquad \frac{\Psi \sqsubseteq \Psi' \qquad i \sqsubseteq i'}{\Psi, i \sqsubseteq \Psi', i'} \qquad \frac{\Psi \sqsubseteq \Psi' \qquad c \sqsubseteq c'}{\Psi, c \sqsubseteq \Psi', c'}$$

$$\frac{\forall i.\ s_i \sqsubseteq s_i'}{\mathbf{interface}\ C\ \{s_1; \ldots\} \sqsubseteq \mathbf{interface}\ C\ \{s_1'; \ldots\}}$$

$$\frac{\begin{array}{c} c = \mathbf{class}\ C(\tau_1\ f_1, \ldots)\ \mathbf{implements}\ C_1, \ldots\ \{d_1; \ldots\} \\ c' = \mathbf{class}\ C(\tau_1'\ f_1, \ldots)\ \mathbf{implements}\ C_1, \ldots\ \{d_1'; \ldots\} \\ \forall i.\ \tau_i \sqsubseteq \tau_i' \qquad \forall i.\ d_i \sqsubseteq d_i' \end{array}}{c \sqsubseteq c'}$$

Method Optimism $s \sqsubseteq s \qquad d \sqsubseteq d$

$$\frac{\tau \sqsubseteq \tau' \qquad \Gamma \sqsubseteq \Gamma'}{\tau\ m(\Gamma) \sqsubseteq \tau'\ m(\Gamma')} \qquad \frac{s \sqsubseteq s' \qquad e \sqsubseteq e'}{s \mapsto e \sqsubseteq s' \mapsto e'}$$

Figure 5.9: Optimism Relation, a.k.a. Precision Relation [Garcia, Clark, and Tanter, 2016; Siek, Vitousek, Cimini, and Boyland, 2015]

Our second theorem states that if a program results in a valuation, then a more optimistic version of that program results in the same valuation or some more optimistic one *unless* the valuation was an overly pessimistic cast in the more pessimistic program.

Theorem 5.7.3 (Gradual Preservation). *For every* Ψ, Ψ', e, e', τ, *and* τ' *such that* $\vdash \Psi$, $\vdash \Psi'$, $\Psi \vdash \tau$, $\Psi' \vdash \tau'$, $\Psi \vdash e \triangleleft \tau$, *and* $\Psi' \vdash e' \triangleleft \tau'$ *hold,*

$$\forall \nu. \begin{pmatrix} \Psi \vdash e \rightsquigarrow^{\infty} \nu : \tau \\ \Psi \sqsubseteq \Psi' \\ \tau \sqsubseteq \tau' \\ e \sqsubseteq e' \end{pmatrix} \implies \exists \nu'. \begin{pmatrix} \Psi' \vdash e' \rightsquigarrow^{\infty} \nu' : \tau' \\ \nu \sqsubseteq \nu' \end{pmatrix} \text{ or } \Psi \vdash \nu \ \texttt{bad-cast}$$

Our third theorem states that, if an optimistic program results in a valuation, then a more pessimistic version results in that same valuation or some more pessimistic one *unless* it encounters an overly pessimistic cast first.

Theorem 5.7.4 (Gradual Reflection). *For every* Ψ, Ψ', e, e', τ, *and* τ' *such that* $\vdash \Psi$, $\vdash \Psi'$, $\Psi \vdash \tau$, $\Psi' \vdash \tau'$, $\Psi \vdash e \triangleleft \tau$, *and* $\Psi' \vdash e' \triangleleft \tau'$ *hold,*

$$\forall \nu'. \begin{pmatrix} \Psi \sqsubseteq \Psi' \\ \tau \sqsubseteq \tau' \\ e \sqsubseteq e' \\ \Psi' \vdash e' \rightsquigarrow^{\infty} \nu' : \tau' \end{pmatrix} \implies \exists \nu. \Psi \vdash e \rightsquigarrow^{\infty} \nu : \tau \wedge \begin{array}{c} \nu \sqsubseteq \nu' \\ \text{or} \\ \Psi \vdash \nu \ \texttt{bad-cast} \end{array}$$

Together, these theorems prove our calculus provides the gradual guarantee. Interestingly, gradual preservation and reflection can be derived from our earlier theorems by making one key observation: making a pro-

gram more optimistic has the effect of making it more likely to be able to reduce pessimistically. Thus our direct semantics provides new perspective on the gradual guarantee.

5.7.4 *Transparency*

Lastly, it is easy to prove the following theorem about our optimism relation:

Theorem 5.7.5 (Transparency)**.**

$$\forall v, v'. \quad v \sqsubseteq v' \quad \Longrightarrow \quad v = v'$$

For languages providing the gradual guarantee, we believe this accurately formalizes our concept of transparency. In particular, the combination implies that making a program more optimistic will not affect the values that arise during that program's execution. This is in contrast to calculi like the cast calculus [Siek, Vitousek, Cimini, and Boyland, 2015], in which two values can be related and yet differ due to inserted casts, which are precisely the wrapper functions we actively avoided in order to get the following promising experimental results.

5.8 EXPERIMENTAL EVALUATION

We claim that our approach to gradual typing can be implemented efficiently and avoid the performance pitfalls of gradual typing that Takikawa et al. [2016] described. Here we present an evaluation of our experi-

mental language called Nom. We used benchmarks from two different sources: first, there are two benchmarks from the benchmark suite used by Takikawa et al. [2016], and second, there are five benchmarks that are among those that Vitousek, Swords, and Siek [2017] selected from the official Python benchmark suite [The Python Development Team, 2008] at the time. These serve to evaluate our implementation on two metrics, respectively. The first set of benchmarks tests the overhead that is introduced at the boundaries between typed and untyped code. The second set of benchmarks tests whether type annotations improve the performance of programs, which is a part of our motivation for gradual typing. For comparison with another sound nominally typed language with gradual typing, we also translated the first group of benchmarks to C#, and we present the results of running those translations alongside the others.

5.8.1 *The Experimental Compiler*

Our experimental compiler supports our language Nom that implements the formalized features discussed so far along with mutable state, primitive types, implementation inheritance, overloading, access/visibility modifiers, and static fields and methods. Unlike in our calculus, field accesses and method invocations are not explicitly annotated with a dispatch mode, and Appendix B.1 discusses how Nom addresses the subtleties involved in bridging this gap.

Because dynamic checks are more common with gradual typing, we make some optimizations to the standard implementation for a nominally typed object-oriented language. At compile time, a number is generated

for each class type. An object is represented as its class number followed by its fields. The class number is used to index arrays that provide standard features such as method tables and interface tables, which are used by statically typed method invocations. Each class index is also associated with a flat list of all its supertypes—class hierarchies are usually rather shallow, so scanning these lists for a matching supertype can be expected to be quick. In order to make dynamically typed method invocations efficient, the class number is used to access an array of association lists mapping method identifiers to dispatching methods, each of which employs a statically determined decision tree to determine which overloading to call, if any, based on the types of the arguments. This is essentially an extension of the hybrid-casting technique of Allende, Fabry, and Tanter [2013] in GradualTalk [Allende, Callaú, et al., 2014]. Furthermore, at the call site of each applicable method invocation, we cache the result of method lookup for the three most recent run-time types of the receiver. This is a standard technique for dynamic languages, known as inline caching [Ahn et al., 2014; Deutsch and Schiffman, 1984].

Rather than compiling to assembly, our compiler translates to C, which can then be compiled by a standard C compiler.[3] We use the Boehm-Demers-Weiser conservative garbage collector [Demers et al., 1990].

5.8.2 *Design of Benchmark Programs*

In contrast to work that adds gradual typing to existing programming languages, we do not have access to a large collection of programs written in our language. However, as a first step, all we need is a program that has a

3 For the benchmarks, we use the Microsoft C compiler, set to optimize for speed (`/O2`).

large number of transitions between untyped and typed code, as these are the only possible sources of gradual-typing overhead in our system. Fortunately, the two smallest poorly performing (i.e. more than 100x slowdown) programs in the benchmark suite of Takikawa et al. were also among those with the highest numbers of boundary transitions. These two programs are `sieve` and `snake`. `sieve` implements the sieve of Eratosthenes using streams to determine the 10,000$^{\text{th}}$ prime number. `snake` implements the popular game Snake and runs it using a statically predetermined list of about 55,000 moves and events. Note that `sieve` in particular was written "to illustrate the pitfalls of sound gradual typing" [Takikawa et al., 2016], as it consists of just two heavily interacting modules.

Given that the programs were written in a different programming paradigm, there are some design choices to be made in how to translate them to Nom and C#. We strove to mimic the structure of the original programs as much as possible in order to keep the numbers and kinds of transitions across module boundaries the same. The biggest differences are that we manually implement tail-recursion elimination in our translation and—as Nom does not support anonymous functions—we model function types using interfaces and closures using classes. All in all, the converted programs are nominal but not necessarily written in an object-oriented style. Thus good performance with these programs is likely to imply good performance in most cases both because they have already been demonstrated to cause problems for prior work due to frequent interaction between modules and because they are written in a style that is not favored by our implementation.

The Python benchmarks were much easier to translate, as they were written in a style that fits our language much more closely. As such, they are more what a typical benchmark for our language would look like.

5.8.3 *Benchmark Results*

All benchmarks were run on an Intel Core i7-3770 at 3.4Ghz with 16GB of main memory, running Windows 7 with minimal background activity. The benchmark programs were run over several iterations. For each iteration, the sequence in which individual configurations were run was determined randomly.

5.8.3.1 *Sieve*

`sieve` is an extreme microbenchmark, consisting of just two heavily interacting modules with several hundred million transitions between those two modules. As such, it is a key benchmark to measure the efficiency of casts in a gradually typed language. The left-hand side of Figure 5.10 shows the results for the `sieve` benchmark for Racket, C#, and Nom. There are four configurations, corresponding to the fully untyped program "00", the fully typed program "11", and the two mixed configurations "01" and "10". In Typed Racket, the two mixed configurations cause extreme overheads due to gradual typing, as described by Takikawa et al. [2016]. C#, on the other hand, is unaffected by interaction but instead suffers significant slowdown in the presence of dynamic typing.

Regarding Nom, its performance is, in relative terms, fairly constant across the configurations, though there is an increase in performance

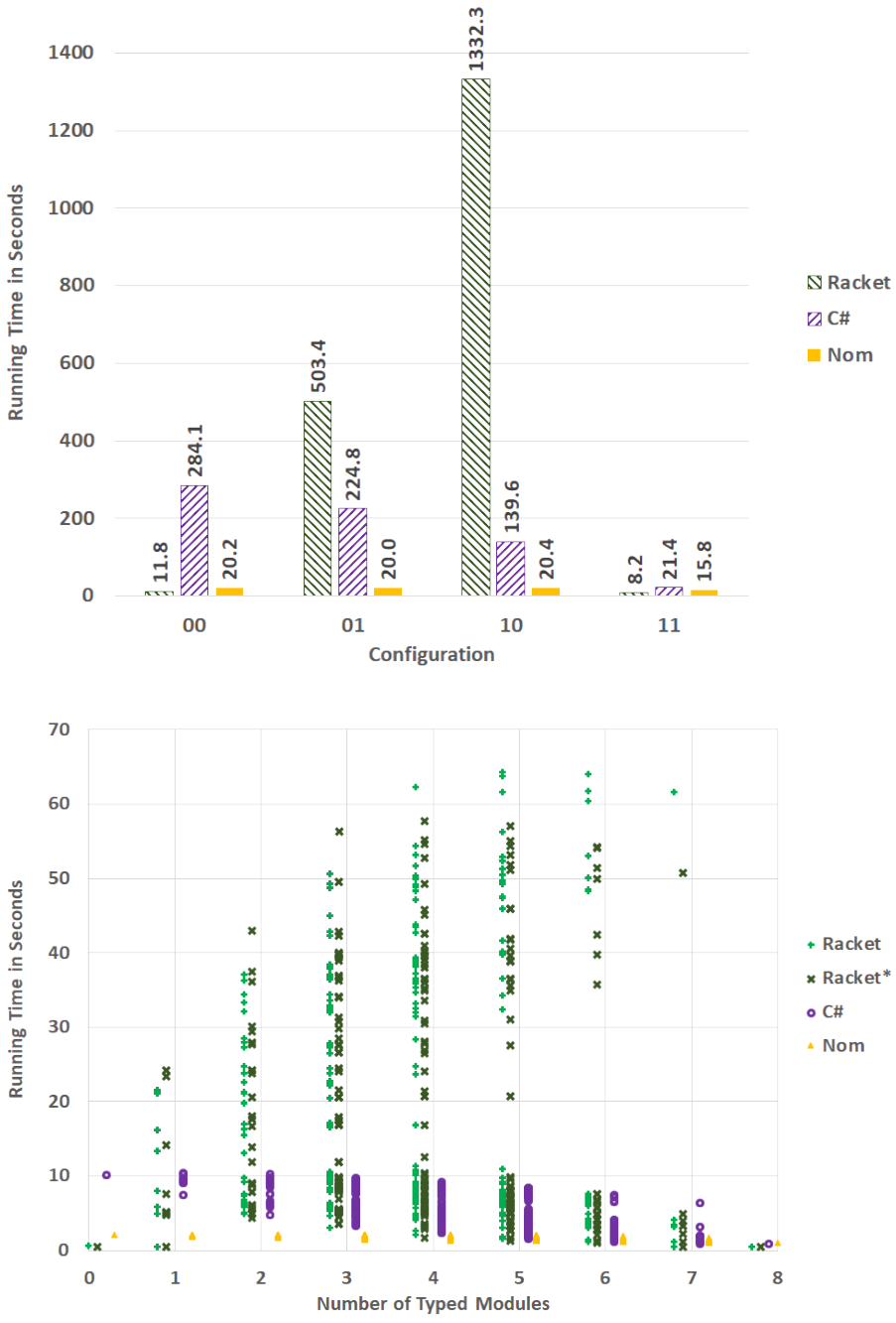


Figure 5.10: Benchmark results for `sieve` (top) and `snake` (bottom)

in the fully typed configuration. This is despite the fact that we still measured several hundreds of millions of transitions between typed and untyped code when executing either mixed configuration in Nom, the same magnitude that Takikawa et al. reported for Racket.

5.8.3.2 *Snake*

The right-hand side of Figure 5.10 shows the timings for `snake` as a scatter plot of running times, in seconds, grouped by the number of typed modules. There are two versions for Racket here because the original version published by Takikawa et al. checks the entire contents of lists when casting them from untyped code to typed code, an operation that in theory increases the time complexity of the programs. We thus developed a modified version of `snake`, labeled Racket*, that uses a user-defined structure instead of Racket's `cons`-lists in an attempt to make those checks lazy, similar to how our Nom implementation of lists works. Interestingly, there does not seem to be much difference in performance between the two Racket versions, suggesting that the performance issues Takikawa et al. observed are due to the concerns we have discussed throughout the chapter rather than due to the use of deep casts. As before, the performance of Nom, on the other hand, consistently improves as more types are added to the program. The same holds for C#, though again suffering significantly more overhead in the presence untyped code.

5.8.3.3 *Python Benchmarks*

For the Python benchmarks, we chose five with some preference towards the ones that had poor performance under the *transient*-cast implementation of Vitousek, Swords, and Siek [2017] (`pystone` and `float` suffer

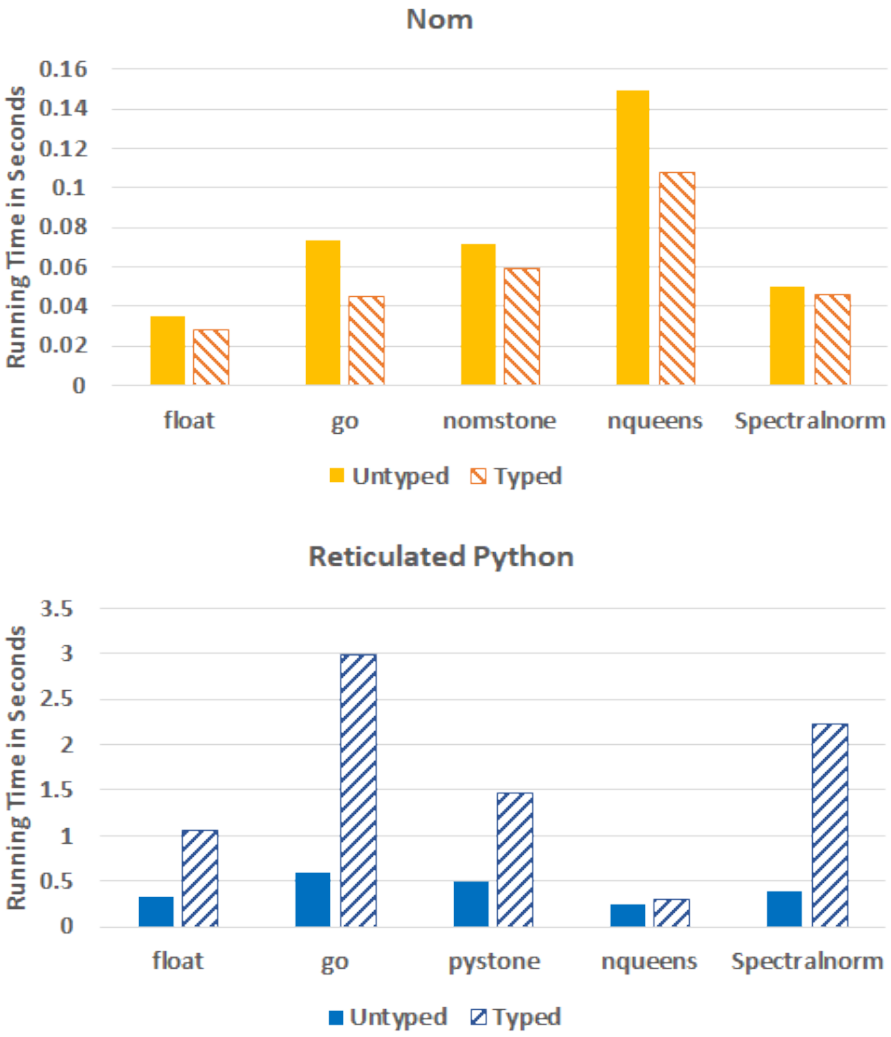


Figure 5.11: Benchmarks taken from Vitousek, Swords, and Siek [2017]'s selection of Python benchmarks

from about 200% overhead, and `go` and `spectralnorm` suffer from about 400% overhead for *typed* code compared to untyped code[4]). In contrast to the Racket benchmarks, these programs were written in a language that is close to ours and thus were translated with minimal effort. The left-hand side of Figure 5.11 shows the results of these benchmarks for Reticulated Python, and the right-hand side of Figure 5.11 shows the results for Nom. The absolute running times should not be compared other than to serve as an indicator of overall reasonableness; Python is interpreted, whereas our code is compiled and optimized by a C compiler, so absolute differences are not meaningful. The effect of types on performance within each language is meaningfully different, though. The transient casting strategy slows down programs as more type annotations are added because type annotations cause checks to be inserted and executed regardless of whether the whole program is typed or not. This may be a reasonable thing to do in the scenarios that Vitousek et al. are considering, where an open world can readily circumvent invariants of the gradual-typing implementation, but we believe that in general programs should overall become faster as more type annotations are added due to the additional optimizations this enables. Nom achieves this goal here, although the `snake` benchmark best illustrates this behavior because it provides data on many intermediate configurations as well.

5.8.4 *Validity*

We only evaluated our system on a small set of small programs. While our system performed well for these programs, there is always the possibility

4 Without blame. Adding blame tracking in many cases more than doubles the overhead.

that it may perform poorly for some other program. However, by the nature of our implementation, our overhead is proportional to the number of run-time interactions between typed and untyped code. Importantly, our overhead is fairly unaffected by the kind of interactions that occur due to the transparency of our casts. Consequently, it is likely the case that `sieve` does in fact represent a worst-case scenario regarding overhead created by our system due to the immense degree of interaction points as designed by Takikawa et al. While it seems possible that there are other programs that could increase our overhead by small factors, say by designing a program to thwart any effectiveness of inline caches, it seems unlikely that there are programs that would increase our overhead by large factors, especially to the degree observed in the related works we compare to.

As for the measurements we do provide, the usual caveats to experimental running-time measurements apply. Efforts we took to mitigate the risk of obtaining misleading numbers include

- running the benchmarks sequentially, not in parallel, with a separate randomized order for each trial run,
- confirming that several minor variations of the Nom benchmarks, such as employing object-oriented-style dynamic dispatch instead of functional-style static dispatch, exhibited similar performances,
- and observing no significant differences in performance across three different machines.

Furthermore, the artifact-evaluation committee approved the validity of our manual translations of the benchmarks and successfully replicated our results.

5.9 SUMMARY

This chapter presented new properties of gradual type systems that, in conjunction with the gradual guarantee, capture an intuition about when and where gradual typing can produce overhead even in the ideal case. The properties do not necessarily guarantee an efficient implementation of gradual typing, as we demonstrate with benchmarks for C#.

We showed, however, that by codesigning the type system and underlying runtime system alongside these desired properties for gradual typing, we could create an efficient and well-behaved gradually typed nominal object-oriented language. We provided evidence that our language does not suffer from previously measured extreme overheads due to gradual typing, even in adversarial scenarios where programs have a high level of interaction between typed and untyped code.

As part of our design, we chose to use nominal typing instead of structural typing as an explicit tradeoff of expressiveness for performance. We argued how this loss of expressiveness is acceptable for many applications of gradual typing, and we illustrated paths forward towards recovering expressiveness while still maintaining performance. In general, there are many desirable features that our language does not have, but it seems that many of them can be added with reasonable effort. As such Nom forms a good foundation from which to explore adding more features while keeping efficiency and good theoretical properties, which we make use of in the next chapter.

6

TRANSITIONING FROM STRUCTURAL TO NOMINAL CODE

6.1 INTRODUCTION

In this chapter, we relax some of the restrictions we accepted in Nom to obtain efficiency, and in doing so, we address another issue, namely that typed and untyped code often exhibits different patterns. Some research has investigated gradually giving types to code exhibiting typical untyped patterns [Allende, Callaú, et al., 2014; Richards, Nardelli, and Vitek, 2015; Siek and Taha, 2007; Swamy et al., 2014; Tobin-Hochstadt and Felleisen, 2008; Vitousek, Kent, et al., 2014; Vitousek, Swords, and Siek, 2017], while other research has investigated gradually removing types from code exhibiting typical typed patterns [Bierman, Meijer, and Torgersen, 2010; Cimini and Siek, 2016; Garcia, Clark, and Tanter, 2016; Ina and Igarashi, 2011; Muehlboeck and Tate, 2017b; Siek and Taha, 2006]. Here we investigate how to extend the gradual-typing concepts so that one can give formal guarantees not only about how types can change as code evolves but also about how such patterns can change as well.

We again do so specifically in the setting of object-oriented languages. Untyped object-oriented languages typically encourage structural patterns whereas typed object-oriented typically encourage nominal patterns. So as code stabilizes, one would expect programmers not only to introduce types

describing the expected behavior of their programs but also to replace structural records using dynamic dictionaries with nominal classes using fields and methods.

Beyond code evolution, some programming tasks are simply better suited for untyped/structural patterns than for typed/nominal patterns, and yet this structural code still needs to interact with nominal code. Test code often has an intimate knowledge of the dynamics of the code being tested and consequently omits methods that are expected by the interaction interface but which are unnecessary for the test at hand. Deserialization code often converts dictionary-like structures such as JSON trees into the nominal structures expected by the program's core. Reflection code treats nominal structures as dictionary-like structures in order to reason about and manipulate them more mechanically.

Whatever the purpose, there are many applications for principled interaction between not just typed and untyped code but also nominal and structural code. As such, Thorn [Wrigstad et al., 2010] explores both forms of mixing. However, its *like* types only provide a way for nominal values to be treated as structural values, making the communication between these two patterns primarily one-sided. This accommodates applications like deserialization, effectively converting structural data into nominal data, and like reflection, manipulating nominal values as structural entities, but not applications like testing, in which structural values need to masquerade as nominal entities. It also fails to provide simple conveniences like unannotated lambdas, which have proven useful enough to even be incorporated into major typed object-oriented languages despite the significant complexity required of the type checker to retrofit such functionality.

A major reason for this is efficiency. Nom demonstrated that having every value know its most precise type at run time enables gradual typing to be implemented efficiently so that increasing type annotations reliably improves program performance as one would expect. However, in order for every value to know its most precise type, Nom requires every lambda expression to be explicitly annotated with the nominal interface it is intended to implement, just like Thorn.

In this chapter we provide a system in which nominal values can masquerade as structural entities and structural values can masquerade as nominal *interfaces* while still providing good performance and semantic guarantees as to how code can evolve from dynamic structures to nominal types. Classes and interfaces can be entirely removed from programs, replacing their nominal components with their structural counterparts throughout the program, and the only significant change in behavior is the slowdown one would expect from switching typed field/method-offset lookups to untyped dictionary-entry lookups. This even works when the classes that are removed implement interfaces that still remain, meaning the records and lambdas replacing class instances need to dynamically discover and adapt to the interfaces that had been implemented by those instances.

6.2 MOTIVATION

Suppose you work at a video-game studio. Video-game development has the interesting characteristic of creative artistic exploration needing to be done simultaneously with high-performance implementation. As

such, video games often use two interacting programming languages, with designers operating in a high-level scripting language and implementers operating in a low-level performance language (although not too low-level in the case of mobile development). Furthermore, as designs stabilize, the implementations of various game mechanics are moved from the high-level language to the low-level language in order to meet the high-performance expectations of players. Unfortunately, each such move is a substantial effort because the two languages are often substantially different and the communication interface between them itself often requires manual implementation effort to provide.

This is an ideal application for gradual typing, but it requires providing a migration path and interaction protocol for more than just types. To see why, consider how the design of characters changes over time. At first, characters might just start with a notion of health, but as time progresses the designers realize the need for more and more attributes such as energy, strength, speed, and so on. And as time progresses further, some of these attributes may be removed or merged for the sake of reducing cognitive load on the player. Designers are used to this ever-changing landscape and employ structural patterns in order to keep up with and adapt to these changes. But once a design a solidifies, it makes sense to coalesce the various attributes of a character into a class and to coalesce the various customizable interactions characters are expected to provide into an interface implemented by that class. And then it makes sense to move that class and interface into the low-level language, along with various scripts for implementing character functionality, to provide better performance, while still providing the scripting language with the necessary hooks for interacting with this character class and specifying the custom interactions

expected by its implemented interface. Thus, in addition to refining type information, this application needs a path for changing from structural code to nominal code that still enables structural code to access and update nominal data and to specify implementations for nominal interfaces.

Of course, efficiency is also important in this application. Note that it is never the case that all code is untyped or all code is typed. Instead, it always the case that some code is untyped but most code is typed. Furthermore, while there needs to be back-and-forth interactions between typed and untyped code, not all interactions are expected to work. In particular, designers do not expect to be able to give an object whose dictionary happens to have all the right entries to typed code that expects specifically instances of a particular class; instead, they would allocate an instance of the class and, in their untyped scripting language, fill its fields with the values they want it to have. In this work we will optimize for when code is mostly typed and rely on these concessions on expected interactions in order to implement (sound) gradual typing efficiently.

Our strategy at a high level is to treat data differently than interactions. In particular, while untyped code will be able to manipulate instances of nominal classes structurally (albeit in a limited fashion in order to maintain the invariants of the class), we will not allow structural data to be implicitly cast to a nominal *class* type. On the other hand, we will allow structural values to be implicitly cast to a nominal *interface* type so that typed nominal code can interact with structural operations just as if they had been instances of classes implementing the expected interface. When such a structural value crosses the boundary, we will *monotonically* retrofit an appropriate v-table and interface table onto the value. Thus typed code will be able to invoke methods on these structural values just

as they would class instances. And, since structural data is not implicitly cast into class instances, typed code can always access fields efficiently. Similarly, when classes are compiled we provide a pointer to a generated field-and-method-offset dictionary in the v-table that untyped code can use to access and mutate fields and invoke methods relatively efficiently. Thus we repurpose the indirections already present in the implementations of both nominal and structural code in order to make values from one setting appear to be values from the other.

Of course there are many high-level and low-level challenges to providing formal guarantees about code evolution and practical assurances about code performance. Understanding these challenges requires a more detailed understanding of the language we provide, so next we provide a calculus containing the features we found were most critical in formulating the challenge and informing our design.

6.3 THE CALCULUS

The calculus is comprised of both structural and nominal features. However, the structural features only exist at the expression level whereas some nominal features only exist at the top level. These nominal features are used to describe the overarching architecture of the program that we must work within. As such, we first present the nominal hierarchy of our calculus.

	Interface Name I	Class Name C
	Field/Method Name f	Variable Name x
Interface	$\mathcal{I}$	$::= \texttt{interface}\ I\langle\Theta\rangle\ \texttt{extends}\ I\langle\sigma,...\rangle,...\ \{ms;...\}$
Class	$\mathcal{C}$	$::= \texttt{class}\ x : C\langle\Theta\rangle(f:\tau,...;\Gamma)$ $\texttt{implements}\ I\langle\sigma,...\rangle,...\ \{ms \mapsto e;...\}$
Material	$\mathcal{M}$	$::= \mathcal{I} \mid \mathcal{C}$
Hierarchy	$\mathcal{H}$	$::= \mathcal{M};...$
Method Name	m	$::= f \mid \lambda$
Method Signature	s	$::= \langle\Theta\rangle(\Gamma) : \tau$
Type Context	Γ	$::= x : \tau,...$
Kind Context	Θ	$::= \alpha,...$
Material Name	M	$::= I \mid C$
(Gradual) Type	τ	$::= \alpha \mid M\langle\tau,...\rangle \mid \mathbb{B} \mid \mathbb{Z} \mid \mathbb{R} \mid \texttt{dyn}$
Concrete Type	σ	$::= \alpha \mid M\langle\sigma,...\rangle \mid \mathbb{B} \mid \mathbb{Z} \mid \mathbb{R}$

Figure 6.1: Grammar of Nominal Hierarchy

6.3.1 *Hierarchy*

The hierarchy of our calculus, presented in Figure 6.1, is comprised of interfaces and classes. The first thing to notice is that our interfaces and classes have type parameters, i.e. are generic. The goal of this work is *not* to design gradual typing for generics, unlike [Ina and Igarashi, 2011], and as such our calculus has no support for important features such as variance. However, even without variance type parameterization imposes significant constraints on efficiently implementing gradual typing that the reader should be aware of. For one, it makes casts more complex—unlike Nom, our casts check a simple nominal tag *and* compare type arguments. For another, it means that an interface imposed upon a structural value might need to be refined as the program proceeds, say do the value first

being cast to $\mathsf{Fun}\langle \mathbf{dyn}, \mathbf{dyn} \rangle$ and later to $\mathsf{Fun}\langle \mathbb{Z}, \mathbb{Z} \rangle$. Furthermore, when monotonically refining the interface imposed upon a structural value, we must be mindful of concurrency issues. Although our calculus is single-threaded, our implementation has made sure that mutating casts are lock free and yet thread safe even under common weak memory models while also ensuring that successful casts can proceed without synchronizing.

Generics also disallow certain tempting solutions. In particular, rather than implementing gradual typing by casting structural *values*, one could implement gradual typing by instead casting the structural *code locations* generating those values. That is, every closure is generated by some lambda expression, and one could monotonically update the type of the lambda expression, essentially dynamically inferring the type annotations for that code location. Such a solution would even satisfy the gradual guarantee [Siek, Vitousek, Cimini, and Boyland, 2015]. However, generics invalidate this solution by asserting that even code locations should be able to take on multiple types (parameteterized by the type variables in the context) as the program executes.

The second thing to notice is that, while classes and interfaces can each inherit multiple interfaces, our calculus does not have class inheritance. This is merely because we found class inheritance to be uninteresting given the design decisions we have already argued, namely that structural values should not be castable to class instances. Our implementation does support subclassing, and our formalism can easily be extended to support subclassing as well. In particular, even when the receiver of a method invocation is known to be an instance of a class, the method is dispatched dynamically rather than statically in case it has been overridden in some subclass.

The final thing to be aware of, although not visible in the grammar, is that our calculus enforces single-instantiation inheritance. This is necessary if we want to be able to disallow a closure that has already been cast to $\mathsf{Fun}\langle\mathbb{Z}, \mathbb{R}\rangle$ from also being cast to the incompatible type $\mathsf{Fun}\langle\mathbb{R}, \mathbb{Z}\rangle$. If classes could implement the same generic interface with multiple instantiations, and if we want lambda expressions to be convertible into class-instance allocations, then the gradual guarantee would require us to support such incompatible casts. In order to enforce this, as well as to make casts more efficient, type arguments to inherited interfaces must be "concrete" types σ rather than (gradual) types τ.

6.3.2 *Fields and Methods*

In order to support structural patterns, we felt it was important for fields and methods to share the same namespace f (which we did decide to distinguish from local variables x). The one exception is the special method name λ. Structural values and instances of classes (and interfaces) with such a method can be invoked as if they were themselves functions. Originally we had intended to treat any interface with a single method as a "functional" interface. However, this means that closures cast to such an interface would become endowed with a method of that single name, which then would need to be invokable even from untyped code in order to support the gradual guarantee. The consequence of this would be that a cast could give a value additional functionality, which is fundamentally incompatible with the gradual guarantee that asserts that removing casts (e.g. making code more dynamic) should not reduce functionality. Thus we

realized it was either necessary to either have some specially designated name such as `invoke` or some non-name symbol such as λ, and we opted for the latter.

Note that methods in our calculus can be parameterized. In particular, this means that if a method were fetched by untyped code as if it were a field, the type of the resulting value is not expressible in our language. We also chose to permit a limited form of overloading where classes and interfaces can provide multiple signatures and implementations for a method provided they all have a different number of type parameters and/or program parameters. In order to avoid complications with LLVM, we did not however opt to permit variadic overloading.

One should notice that, just after the declaration of fields, classes also declare a context Γ of private data. This data intentionally has no field associated with it so that it cannot be accessed through any means besides the method implementations. We use so that class instances can replace closures without exposing any new functionality; the context Γ is the collection of local variables captured by the closure.

6.3.3 *Types*

The types in our language include type variables α, parameterized nominal interfaces and classes $M\langle\tau, ...\rangle$, primitive booleans $\mathbb{B}$, (64-bit) integers $\mathbb{Z}$, and (64-bit IEEE 764 floating point) reals $\mathbb{R}$, and the "dynamic" type **`dyn`**. Note that we do not have a function type $\tau \to \tau$. This actually makes our challenge *more* difficult, rather than easier, because instead we need to support lambdas through nominal interfaces with a λ method, and there

can be multiple such interfaces within a given program *and* a lambda must be simultaneously castable to multiple such interfaces since classes can implement multiple interfaces.

For the purposes of this chapter, it is important to stress that we intentionally take a broad perspective of the **dyn** type. We view **dyn** as a means to explicitly circumvent the type system. That is, rather than **dyn** being just a type that has yet to be determined, we also view **dyn** as potentially conveying the programmer's intent to reason about the *dynamics* of the program beyond what the type system is capable of expressing. For example, **dyn** can represent values that are untypeable in our calculus, such as structural values. Also, **dyn** can be used to access functionality in a value not explicitly accessible by its type. For example, a closure can be cast to $\mathsf{Fun}\langle\mathbb{Z},\mathbb{Z}\rangle$ and then back to **dyn** and, unlike most (sound) gradually typed systems, our calculus permits the resulting cast value to still be supplied, say, $\mathbb{R}$s and to return $\mathbb{R}$s when invoked from untyped code with the understanding that the programmer may be well aware that the dynamics of the program guarantees that this particular function also operates on other numical values even if the types are unaware of or unable to express the fact.

6.3.4 *Expressions*

The grammar of our expressions is shown in Figure 6.2, divided into five rows. The first row specifies the basic components. The second and third rows contain the three ways to create non-primitive values. The fourth row describes the operations on those non-primitive values. The final row is not

$$
\begin{array}{lll}
\multicolumn{3}{l}{\text{Boolean } b \in \mathbb{B} = \{\texttt{false}, \texttt{true}\} \qquad \text{Integer } i \in \mathbb{Z} \qquad \text{Real } r \in \mathbb{R}} \\
\text{Expression} & e ::= x \mid \texttt{let}\ x : \tau := e\ \texttt{in}\ e \mid b \mid i \mid r \mid e + e & \\
& \quad \mid \texttt{new}\ C\langle\sigma, ...\rangle(e, ...; e, ...) \mid \texttt{new}\ \lambda s \mapsto e & \\
& \quad \mid \texttt{new}\ (f := e; ...)_x\{ms \mapsto e; ...\} & \\
& \quad \mid e.f \mid e.f := e \mid e\langle\sigma, ...\rangle(e, ...)^\delta \mid e \approx e & \\
& \quad \mid \ell \mid \texttt{cast}\ e\ \texttt{to}\ \tau \mid \texttt{cast}\ e\ \texttt{to}\ \tau\ \texttt{for}\ \ell.m\langle\sigma, ...\rangle(\nu, ...) & \\
\text{Dispatch Mode} & \delta ::= M.m \mid \texttt{dyn} &
\end{array}
$$

Figure 6.2: Grammar of Expressions

part of the surface language; it is only used for specifying the semantics, and so we defer its discussion to Section 6.6.

6.3.4.1 *Primitives*

Our calculus has three primitive values: booleans b, integers i, and reals r. For booleans, our language provides standard operations such as $\wedge$ and $\vee$ as well as standard control flow such as if-then-else and while constructs, but we elide these from the calculus solely because they proved to complicate the formalism (say by requiring union types) without contributing meaningful insights. For integers and reals, our calculus provides addition (which can operate on any combination of integers and reals), and our language provides many other standard arithmetic operations.

6.3.4.2 *Instances, Records, and Closures*

Our calculus provides expressions for allocating (on the heap) class instances, structural records, and lambda closures. These are all closely related. In particular, class instances are the nominal counterpart to both structural records and lambda closures. Lambda closures are essentially

structural records with a single λ method, but they have a more optimized implementation for this common use case and a more eager cast semantics in order to make that optimization accessible even when lambda closures cross the boundary into typed nominal code.

The expression **new** $C\langle\sigma_1, ...\rangle(e_{f;1}, ...; e_{x;1}, ...)$ allocates an instance of the class C with concrete type arguments $\sigma_1, ...$, with values for its private data given positionally by $e_{x;1}, ...$, and with initial values for fields given positionally by $e_{f;1}, ...$

The expression **new** $(f_1 := e_{f;1}; ...)_x\{m_1 s_1 \mapsto e_{m;1}; ...\}$ allocates a record. The initial values of its initial mutable fields $f_1, ...$ are given by the corresponding expressions $e_{f;1}, ...$ The signatures and implementations of its immutable methods are given by $m_1 s_1 \mapsto e_{m;1}, ...$, which can reference the variable x representing the "this" pointer of the record. New fields can be added to the record (though, for simplicity, not removed) by subsequent field assignments. As such, the use of fields in this expression can be viewed as a shorthand, but we make use of this shorthand so that, if ever this record is converted into a class, the resulting nominal program will reduce in the same order as this structural program (evaluating values for fields before allocating the new reference), making the statement and proof of our main theorem much simpler.

The expression **new** $\lambda s \mapsto e$ allocates a lambda closure. The body e can access local variables in the context (and the same goes for the bodies of methods in structures). This of course is important for implementing lambda expressions, but our calculus will gloss over these implementation details and simply substitute local variables with their values even inside the bodies of lambda expressions.

6.3.4.3 *Field Access and Method Invocation*

Field access and method invocation seem straightforward, but there are two important subtle challenges to be aware of. Consider the expression $e.f\langle\sigma, ...\rangle(...)^{\delta}$. In typed code, we can use the type of e to determine whether this an invocation of the method f on e or an invocation of a functional interface (i.e. λ method) on the result of the field access $e.f$. In untyped code, we cannot make any such distinction at compile time and must determine which is the case at run time. This is the first challenge, and in order to keep our calculus simple we chose to address it by having the former case take one step to reduce (combining method lookup and invocation into one) even though the latter case takes two steps. An alternative and more expressive solution that would be relatively easy to implement would be to have the untyped expression $e.f$ reduce to a method-with-receiver closure that can then be invoked directly and/or cast to a functional interface just as a lambda closure can be.

The other challenge is, in the face of gradual typing, determining what types the arguments should be cast to. This is the purpose of the "dispatch mode" annotation δ on the invocation. The value of this annotation is uniquely determined by the type of the receiver e, making it essentially a note generated by the compiler recording which semantics and dispatch strategy (e.g. v-table, interface table, or dictionary lookup) should be used for the invocation. If the receiver has a statically known type, then the arguments should be cast to the parameter types of the *most precise* signature for that method. If the receiver has a type **`dyn`**, then the arguments should be cast to the parameter types of the dynamically fetched method implementation.

Note that in the statically typed case, it is important to use the *most precise* signature. Suppose an interface `Sub` has a signature that accepts a $\top$ (which, for the sake of discussion, represents any value) but also inherits an interface `Super` whose corresponding signature accepts only a $\bot$ (which, for the sake of discussion, is uninhabited). Then if the receiver has static type `Sub` it would be incorrect to blame [Wadler and Findler, 2009] the caller once the dynamically typed argument is inevitably determined to not be a $\bot$. On the other hand, if the receiver has static type `Super`, then the caller *should* be blamed *even if* the receiver's dynamic type turns out to be `Sub`. Thus the most precise *static* type of the receiver in fact affects the semantics of the invocation, and so we need the dispatch mode δ to record which semantics to use even after the *dynamic* type of the receiver is determined.

6.3.4.4 *Equality*

Finally (since we are deferring discussion of the fourth row to Section 6.6), we close with the equality operator $\approx$. This operator checks that the two values are identical (rejecting even $1 \approx 1.0$). Most importantly, this includes values that are references to the heap. This means that, in order to preserve program behavior as (correct) type annotations are introduced, casts must not introduce references. We want the operator $\approx$ to also have the property that returning true guarantees identical behavior, i.e. $\texttt{if}\ x \approx x'\ \texttt{then}\ e_t$ else e_f should be semantically equivalent to $\texttt{if}\ x \approx x'\ \texttt{then}\ e_t[x' \mapsto x]$ else e_f. This means that casts cannot even "chaperone" references as is done in Typed Racket [Tobin-Hochstadt and Felleisen, 2008]. Although we will employ monotonic casts by choice, we suspect that these requirements actually force that choice (or some less-

sound [Greenman and Felleisen, 2018] semantics like transient casts [Vitousek, Kent, et al., 2014; Vitousek, Swords, and Siek, 2017]).

6.4 THE TYPE SYSTEM

The focus of this chapter is not type systems but rather transitioning code from structural to nominal patterns. Nonetheless gradual typing is critical to enabling this transition. Here we present the type system of our calculus in brief, just highlighting the components that are most relevant to our goal.

6.4.1 *Precision, Inheritance, and Subtyping*

Expressions aside for now, there are many interesting relationships between types alone due to the combination of gradual typing and inheritance. That is, one can adjust whether inheritance is used and how dynamism is introduced or eliminated, and each of these combinations has some interesting utility.

If we ignore inheritance and just focus on dynamism, then we get the consistency ($\sim$) and precision ($\preccurlyeq$) relations (the latter of which is traditionally denoted $\sqsubseteq$, but we reserve that symbol for another notion of precision). The consistency relation holds when there is some way to instantiate occurrences of **`dyn`** in both types in order to make the two types identical. The precision relation holds when there is some way to instantiate occurrences of **`dyn`** in only the right-hand type in order to make

the two types identical. That is, $\tau \preccurlyeq \tau'$ holds when τ' is "more dynamic" than τ.

If we instead ignore dynamism and focus on inheritance, then we get the inheritance relation ($\leq$). The inheritance relation holds when either the two types are the same or the class or interface type on the left can be repeatedly replaced with some inherited interface type to arrive at exactly the type on the right.

Lastly, we can combine both dynamism and inheritance. Pessimistic subtyping ($\blacktriangleleft$) has the property that any value satisfying the contract of the left-hand type is guaranteed to satisfy the contract of the right-hand type. Optimistic subtyping ($\triangleleft$) conceptually (although not precisely) holds whenever there is some way to instantiate occurrences of **`dyn`** in both types in order to make the left inherit the right. Substitutive subtyping ($<:$) has the property that whenever $\tau <: \tau'$ holds then any expression of type τ can replace any expression of type τ' in a well-typed program and the result will still be well-typed.

6.4.2 *Expressions*

The judgements and rules for expression typing are presented in Figure 6.4.The first thing to notice is that there are two typing rules for judgements: $\mathcal{H} \mid \Theta \mid \Gamma \vdash e \triangleleft \tau$ and $\mathcal{H} \mid \Theta \mid \Gamma \vdash e : \tau$. The former judgement should be read as "*e optimistically* has type τ" (in the relevant context). This is the judgement that is actually used to determine whether an expression can be used at a particular location in the program, as can be seen in, say, the rule for typing `let` expressions. The latter judgement should be

Precision $\tau \sim \tau \qquad \tau \preccurlyeq \tau$

$$\frac{}{\tau \sim \tau} \qquad \frac{}{\tau \preccurlyeq \tau}$$

$$\frac{}{\mathbf{dyn} \sim \tau} \qquad \frac{}{\tau \sim \mathbf{dyn}} \qquad \frac{}{\tau \preccurlyeq \mathbf{dyn}}$$

$$\frac{\tau_1 \sim \tau_1' \qquad ...}{M\langle \tau_1, ... \rangle \sim M\langle \tau_1', ... \rangle} \qquad \frac{\tau_1 \preccurlyeq \tau_1' \qquad ...}{M\langle \tau_1, ... \rangle \preccurlyeq M\langle \tau_1', ... \rangle}$$

Inheritance $\mathcal{H} \vdash \tau \leq \tau$

$$\frac{}{\mathcal{H} \vdash \tau \leq \tau}$$

$$\frac{\texttt{interface}\ I\langle \alpha_1, ... \rangle\ \texttt{extends}\ I'\langle \sigma_1, ... \rangle \in \mathcal{H} \qquad \mathcal{H} \vdash I'\langle \sigma_1[\alpha_1 \mapsto \tau_1, ...], ... \rangle \leq \tau'}{\mathcal{H} \vdash I\langle \tau_1, ... \rangle \leq \tau'}$$

$$\frac{\texttt{class}\ C\langle \alpha_1, ... \rangle\ \texttt{implements}\ I\langle \sigma_1, ... \rangle \in \mathcal{H} \qquad \mathcal{H} \vdash I\langle \sigma_1[\alpha_1 \mapsto \tau_1, ...], ... \rangle \leq \tau'}{\mathcal{H} \vdash C\langle \tau_1, ... \rangle \leq \tau'}$$

Subtyping
$\tau \blacktriangleleft \tau$ (Pessimistic)
$\tau \triangleleft \tau$ (Optimistic)
$\tau <: \tau$ (Substitutive)

$$\frac{\mathcal{H} \vdash \tau \leq \tau' \qquad \tau' \preccurlyeq \tau''}{\mathcal{H} \vdash \tau \blacktriangleleft \tau''} \qquad \frac{\mathcal{H} \vdash \tau \leq \tau' \qquad \tau' \sim \tau''}{\mathcal{H} \vdash \tau \triangleleft \tau''} \qquad \frac{\mathcal{H} \vdash \tau \leq \tau' \qquad \tau'' \preccurlyeq \tau'}{\mathcal{H} \vdash \tau <: \tau''}$$

Figure 6.3: Precision, Inheritance, and Subtyping

Optimistic Typing $\mathcal{H} \mid \Theta \mid \Gamma \vdash e \triangleleft \tau$

$$\frac{\mathcal{H} \mid \Theta \mid \Gamma \vdash e : \tau_e \qquad \mathcal{H} \vdash \tau_e \triangleleft \tau}{\mathcal{H} \mid \Theta \mid \Gamma \vdash e \triangleleft \tau}$$

Method Validation $\mathcal{H} \mid \Theta \vdash s \mapsto e$ | Signature Validation $\mathcal{H} \mid \Theta \vdash s$

$$\frac{\mathcal{H} \mid \Theta \vdash \langle \Theta' \rangle(\Gamma') : \tau \qquad \mathcal{H} \mid \Theta, \Theta' \mid \Gamma, \Gamma' \vdash e \triangleleft \tau}{\mathcal{H} \mid \Theta \mid \Gamma \vdash \langle \Theta' \rangle(\Gamma') : \tau \mapsto e} \qquad \frac{\mathcal{H} \mid \Theta, \Theta' \vdash \tau_1 \quad \ldots \quad \mathcal{H} \mid \Theta, \Theta' \vdash \tau}{\mathcal{H} \mid \Theta \vdash \langle \Theta' \rangle(x_1 : \tau_1, \ldots) : \tau}$$

Method Lookup $\mathcal{H} \vdash M\langle \tau, \ldots \rangle.ms$

$$\frac{\textbf{class}\ C\langle \alpha_1, \ldots \rangle\ \{\ldots; ms \mapsto e; \ldots\} \in \mathcal{H}}{\mathcal{H} \vdash C\langle \tau_1, \ldots \rangle.ms[\alpha_1 \mapsto \tau_1, \ldots]} \qquad \frac{\textbf{interface}\ I\langle \alpha_1, \ldots \rangle\ \{\ldots; ms; \ldots\} \in \mathcal{H}}{\mathcal{H} \vdash I\langle \tau_1, \ldots \rangle.ms[\alpha_1 \mapsto \tau_1, \ldots]}$$

Method Prototypes $|ms| = \langle m, n, n \rangle$

$$|m\langle \alpha_1, \ldots, \alpha_{n_\alpha} \rangle(x_1 : \tau_1, \ldots, x_{n_x} : \tau_{n_x}) : \tau| = \langle m, n_\alpha, n_x \rangle$$

Primitive Operation Typing $\tau + \tau : \tau$

$$\begin{array}{ccc} \mathbb{Z} + \mathbb{Z} : \mathbb{Z} & \mathbb{Z} + \mathbb{R} : \mathbb{R} & \mathbb{Z} + \textbf{dyn} : \textbf{dyn} \\ \mathbb{R} + \mathbb{Z} : \mathbb{R} & \mathbb{R} + \mathbb{R} : \mathbb{R} & \mathbb{R} + \textbf{dyn} : \textbf{dyn} \\ \textbf{dyn} + \mathbb{Z} : \textbf{dyn} & \textbf{dyn} + \mathbb{R} : \textbf{dyn} & \textbf{dyn} + \textbf{dyn} : \textbf{dyn} \end{array}$$

Expression Typing $\mathcal{H} \mid \Theta \mid \Gamma \vdash e : \tau$

$$\frac{}{\mathcal{H} \mid \Theta \mid \Gamma \vdash i : \mathbb{Z}} \qquad \frac{}{\mathcal{H} \mid \Theta \mid \Gamma \vdash r : \mathbb{R}}$$

$$\frac{\mathcal{H} \mid \Theta \mid \Gamma \vdash e_1 : \tau_1 \qquad \mathcal{H} \mid \Theta \mid \Gamma \vdash e_2 : \tau_2 \qquad \tau_1 + \tau_2 : \tau}{\mathcal{H} \mid \Theta \mid \Gamma \vdash e_1 + e_2 : \tau}$$

$$\frac{\mathcal{H} \mid \Theta \mid \Gamma \vdash e_1 : \tau_1 \qquad \mathcal{H} \mid \Theta \mid \Gamma \vdash e_2 : \tau_2}{\mathcal{H} \mid \Theta \mid \Gamma \vdash e_1 \approx e_2 : \mathbb{B}}$$

Figure 6.4: Expression Typing

Expression Typing (contd.) $\mathcal{H} \mid \Theta \mid \Gamma \vdash e : \tau$

$$\frac{\begin{array}{c}\textbf{class}\ x : C\langle\alpha_1, ...\rangle(f_1 : \tau_{f;1}, ...; x_1 : \tau_{x;1}, ...)\ \textbf{implements}\ ...\ \{...\} \in \mathcal{H} \\ \mathcal{H} \mid \Theta \vdash \sigma_1 \quad ... \quad \mathcal{H} \mid \Theta \mid \Gamma \vdash e_{x;1} \triangleleft \tau_{x;1}[\alpha_1 \mapsto \sigma_1, ...] \quad ... \\ \mathcal{H} \mid \Theta \mid \Gamma \vdash e_{f;1} \triangleleft \tau_{f;1}[\alpha_1 \mapsto \sigma_1, ...] \quad ...\end{array}}{\mathcal{H} \mid \Theta \mid \Gamma \vdash \textbf{new}\ C\langle\sigma_1, ...\rangle(e_{f;1}, ...; e_{x;1}, ...) : C\langle\sigma_1, ...\rangle}$$

$$\frac{\mathcal{H} \mid \Theta \mid \Gamma \vdash s \mapsto e}{\mathcal{H} \mid \Theta \mid \Gamma \vdash \textbf{new}\ \lambda s \mapsto e : \textbf{dyn}}$$

$$\frac{\begin{array}{c}\mathcal{H} \mid \Theta \mid \Gamma \vdash e_1 \triangleleft \textbf{dyn} \quad ... \quad \text{for all } n, n' \text{ if } f_n = f_{n'} \text{ then } n = n' \\ f_1 \notin \{m_1, ...\} \quad ... \quad \text{for all } n, n' \text{ if } |m_n s_n| = |m_{n'} s_{n'}| \text{ then } n = n' \\ \mathcal{H} \mid \Theta \mid \Gamma, x : \textbf{dyn} \vdash s_1 \mapsto e_1 \quad ...\end{array}}{\mathcal{H} \mid \Theta \mid \Gamma \vdash \textbf{new}\ (f_1 := e_1; ...)_x\{m_1 s_1 \mapsto e_1'; ...\} : \textbf{dyn}}$$

$$\frac{\begin{array}{c}\mathcal{H} \mid \Theta \mid \Gamma \vdash e : C\langle\tau_1, ...\rangle \\ \textbf{class}\ x : C\langle\alpha_1, ...\rangle(..., f : \tau, ...; ...)\ \textbf{implements}\ ...\ \{...\} \in \mathcal{H}\end{array}}{\mathcal{H} \mid \Theta \mid \Gamma \vdash e.f : \tau[\alpha_1 \mapsto \tau_1, ...]}$$

$$\frac{\mathcal{H} \mid \Theta \mid \Gamma \vdash e : \textbf{dyn}}{\mathcal{H} \mid \Theta \mid \Gamma \vdash e.f : \textbf{dyn}}$$

$$\frac{\begin{array}{c}\mathcal{H} \mid \Theta \mid \Gamma \vdash e : C\langle\tau_1, ...\rangle \\ \textbf{class}\ x : C\langle\alpha_1, ...\rangle(..., f : \tau, ...; ...)\ \textbf{implements}\ ...\ \{...\} \in \mathcal{H} \\ \mathcal{H} \mid \Theta \mid \Gamma \vdash e_f \triangleleft \tau_1[\alpha_1 \mapsto \sigma_1, ...]\end{array}}{\mathcal{H} \mid \Theta \mid \Gamma \vdash e.f := e_f : C\langle\tau_1, ...\rangle}$$

$$\frac{\mathcal{H} \mid \Theta \mid \Gamma \vdash e : \textbf{dyn} \qquad \mathcal{H} \mid \Theta \mid \Gamma \vdash e_f \triangleleft \textbf{dyn}}{\mathcal{H} \mid \Theta \mid \Gamma \vdash e.f := e_f : \textbf{dyn}}$$

$$\frac{\begin{array}{c}\mathcal{H} \mid \Theta \mid \Gamma \vdash e : M\langle\tau_1, ...\rangle \qquad \mathcal{H} \vdash M\langle\tau_1, ...\rangle.f\langle\alpha_1, ...\rangle(x_1 : \tau_1, ...) : \tau \\ \mathcal{H} \mid \Theta \vdash \sigma_1 \quad ... \quad \mathcal{H} \mid \Theta \mid \Gamma \vdash e_1 \triangleleft \tau_1[\alpha_1 \mapsto \sigma_1, ...] \quad ...\end{array}}{\mathcal{H} \mid \Theta \mid \Gamma \vdash e.f\langle\sigma_1, ...\rangle(e_1, ...)^M : \tau[\alpha_1 \mapsto \sigma_1, ...]}$$

$$\frac{\begin{array}{c}\mathcal{H} \mid \Theta \mid \Gamma \vdash e : M\langle\tau_1, ...\rangle \qquad \mathcal{H} \vdash M\langle\tau_1, ...\rangle.\lambda\langle\alpha_1, ...\rangle(x_1 : \tau_1, ...) : \tau \\ \mathcal{H} \mid \Theta \vdash \sigma_1 \quad ... \quad \mathcal{H} \mid \Theta \mid \Gamma \vdash e_1 \triangleleft \tau_1[\alpha_1 \mapsto \sigma_1, ...] \quad ...\end{array}}{\mathcal{H} \mid \Theta \mid \Gamma \vdash e\langle\sigma_1, ...\rangle(e_1, ...)^M : \tau[\alpha_1 \mapsto \sigma_1, ...]}$$

$$\frac{\mathcal{H} \mid \Theta \mid \Gamma \vdash e : \textbf{dyn} \qquad \mathcal{H} \mid \Theta \vdash \sigma_1 \quad ... \quad \mathcal{H} \mid \Theta \mid \Gamma \vdash e_1 \triangleleft \textbf{dyn} \quad ...}{\mathcal{H} \mid \Theta \mid \Gamma \vdash e\langle\sigma_1, ...\rangle(e_1, ...)^{\textbf{dyn}} : \textbf{dyn}}$$

Figure 6.4 (contd.): Expression Typing (MonNom)

read as "*e precisely* has type τ". This judgement is primarily used as a convenience for optimistic type checking *except* that it is used to determine the appropriate dispatch mode for a particular invocation for the reasons discussed in Section 6.3.4.3.

Most rules are straightforward, so we just discuss the most interesting ones. Class-instance allocations have the obvious type, and record allocations are entirely dynamic, but notice that lambda expressions also have the **dyn** type. That is, lambda expressions do not have any sort of function type, nor are they inferred to have some functional-interface type. These rules emphasize our approach of completely bundling types and nominality—any structural values simply have type **dyn**. Thus, unlike Thorn, we completely rely upon gradual typing for integrating nominal and structure code.

The remaining interesting rules for expression typing are the rules for invocations. Notice that all of these rules use $e : \tau$ rather than $e \triangleleft \tau$ to check the type of the receiver. This is so that the dispatch mode is guaranteed to be determined by the receiver's most precise type. It also ensures that the only dynamism in the return type of the expression comes from the dynamism in the reciever's type and the dynamism in the method's signature. That is, if the receiver's type and the method's signature are completely concrete, then this guarantees that the return type is completely concrete as well.

The first two rules for invocation handle the case where the receiver has a nominal static type. The first rule handles specifically method invocation—the case where f is actually a field with some functional-interface type is handled by the combination of the field-access rule (not shown) and the second invocation rule. Besides the obvious syntactic difference between

the two rules, the premises only differ in that the first looks up the signature of method f whereas the second looks up the signature of method λ.

The third rule for invocation handles the case where the invokee has a dynamic type. There is no rule for specifically method invocation, since it is covered by the combination of the (dynamic) field-access rule and the third invocation rule.

6.4.3 *Classes and Interfaces*

Lastly, we present the typing rules for classes, interfaces, and the nominal hierarchy in Figure 6.5. The first rules type the hierarchy itself and are designed to address the fact that, while inheritance needs to be non-circular, the signatures of many classes and interfaces are mutually recursive. Thus, the judgement $\mathcal{H} \mid \mathcal{H}' \vdash \mathcal{M}$ indicates that the class or interface definition $\mathcal{M}$ is valid in the entire nominal hierarchy $\mathcal{H}$ provided that $\mathcal{H}'$ is the restriction of that hierarchy to classes and interfaces defined "before" $\mathcal{M}$.

The rules for classes and interfaces check a long list of factors: 1) There is no "earlier" class or interface with the same name. 2) In the case of classes, the types of all fields are valid, there are no duplicate fields, and there is no overlap between between field and method names. 3) No two method signatures have the same method name, accept the same number of type arguments, and accept the same number of program arguments. 4) All method signatures are valid and, in the case of classes, their corresponding implementations have the appropriate types. 5) All inherited interfaces are

defined "earlier" and have valid corresponding type arguments. 6) The class or interface does not inherit, directly or indirectly, two distinct instantiations of any interface. 7) All method signatures in the inherited interface are, in the case of interfaces, extended by or, in the case of classes, implemented by some method signature of the class or interface.

This final item of checking extension or implementation of method signatures warrants extra attention. Without gradual typing, its well known that the subclass or subinterface can soundly broaden the input types and narrow the return type. That is, signature overriding is contravariant on inputs and covariant on the output with respect to *subtyping*. But with gradual typing there are multiple notions of subtyping, and it is not necessarily obvious which subtyping should be used. In developing our proofs, we in fact determined that different notions of subtyping are appropriate for different situations.

Consider first the return type. Suppose interface `Super`'s method returns **`dyn`** and subinterface `Sub` overrides that method to return $I\langle\rangle$. Those familiar with gradual typing might at first think this is acceptable, since the type $I\langle\rangle$ is more precise, in the sense of gradual typing, than **`dyn`**. Thus `Sub` is just making the intent of the method more precise. But now suppose one invokes the method on a variable of type `Sub` and one knows that the dynamics of this particular instance of `Sub` guarantees the returned value will be an instance of class $C\langle\rangle$ which implements $I\langle\rangle$. Had the variable had the less informative type `Super`, then the return type would be **`dyn`** and the programmer would be able to directly exploit their understanding of the dynamics. But with the more informative type `Sub`, the programmer must explicitly cast the return type to **`dyn`** to do so. The issue is fundamentally that `Sub`'s return type can be used in settings where `Super`'s cannot.

Hierarchy Typing $\vdash \mathcal{H}$ $\quad \mathcal{H} \vdash \mathcal{H}$

$$\frac{\mathcal{H} \vdash \mathcal{H}}{\vdash \mathcal{H}} \qquad \frac{}{\mathcal{H} \vdash \varnothing} \qquad \frac{\mathcal{H} \vdash \mathcal{H}' \qquad \mathcal{H} \mid \mathcal{H}' \vdash \mathcal{M}}{\mathcal{H} \vdash \mathcal{H}'; \mathcal{M}}$$

Material Typing $\mathcal{H} \mid \mathcal{H} \vdash \mathcal{M}$

$$\frac{\begin{array}{c} I \notin \mathcal{H}' \\ \text{for all } n, n' \text{ if } |m_n s_n| = |m_{n'} s_{n'}| \text{ then } n = n' \qquad \mathcal{H} \mid \Theta \vdash s_1 \qquad \ldots \\ \texttt{interface}\ I_1\langle \alpha_1^1, \ldots \rangle\ \{m_1^1 s_1^1; \ldots\} \in \mathcal{H}' \qquad \ldots \qquad \mathcal{H} \mid \Theta \vdash \sigma_1^1 \qquad \ldots \\ \forall I', \overline{\sigma_1'}, \overline{\sigma_1''}.\ \mathcal{H} \vdash I\langle\Theta\rangle \leq I'\langle\overline{\sigma_1'}\rangle\ \wedge\ \mathcal{H} \vdash I\langle\Theta\rangle \leq I'\langle\overline{\sigma_1''}\rangle \implies \overline{\sigma_1'} = \overline{\sigma_1''} \\ \forall m's' \in \{m_1^1 s_1^1[\alpha_1^1 \mapsto \sigma_1^1, \ldots], \ldots\}.\ \exists ms \in \{m_1 s_1, \ldots\}.\ \mathcal{H} \vdash ms\ \texttt{extends}\ m's' \end{array}}{\mathcal{H} \mid \mathcal{H}' \vdash \texttt{interface}\ I\langle\Theta\rangle\ \texttt{extends}\ I_1\langle\sigma_1^1, \ldots\rangle, \ldots\ \{m_1 s_1; \ldots\}}$$

$$\frac{\begin{array}{c} C \notin \mathcal{H}' \\ \mathcal{H} \mid \Theta \vdash \Gamma \qquad \mathcal{H} \mid \Theta \vdash \tau_1 \qquad \ldots \qquad \text{for all } n, n' \text{ if } f_n = f_{n'} \text{ then } n = n' \\ f_1 \notin \{m_1, \ldots\} \qquad \ldots \qquad \text{for all } n, n' \text{ if } |m_n s_n| = |m_{n'} s_{n'}| \text{ then } n = n' \\ \mathcal{H} \mid \Theta \mid x : C\langle\Theta\rangle, \Gamma \vdash s_1 \mapsto e_1 \qquad \ldots \\ \texttt{interface}\ I_1\langle \alpha_1^1, \ldots \rangle\ \{m_1^1 s_1^1; \ldots\} \in \mathcal{H}' \qquad \ldots \qquad \mathcal{H} \mid \Theta \vdash \sigma_1^1 \qquad \ldots \\ \forall I', \overline{\sigma_1'}, \overline{\sigma_1''}.\ \mathcal{H} \vdash C\langle\Theta\rangle \leq I'\langle\overline{\sigma_1'}\rangle\ \wedge\ \mathcal{H} \vdash C\langle\Theta\rangle \leq I'\langle\overline{\sigma_1''}\rangle \implies \overline{\sigma_1'} = \overline{\sigma_1''} \\ \forall m_I s_I \in \{m_1^1 s_1^1[\overline{\alpha_1^1 \mapsto \sigma_1^1}]\}.\ \exists m_C s_C \in \{\overline{m_1 s_1}\}.\ \mathcal{H} \vdash m_C s_C\ \texttt{implements}\ m_I s_I \end{array}}{\mathcal{H} \mid \mathcal{H}' \vdash \texttt{class}\ x : C\langle\Theta\rangle(\overline{f_1 : \tau_1}; \Gamma)\ \texttt{implements}\ \overline{I_1\langle\overline{\sigma_1^1}\rangle}\ \{\overline{m_1 s_1 \mapsto e_1}\}}$$

Method overriding $\mathcal{H} \vdash ms\ \texttt{extends}\ ms$ $\quad \mathcal{H} \vdash ms\ \texttt{implements}\ ms$

$$\frac{\mathcal{H} \vdash \tau_1' \leq \tau_1 \qquad \ldots \qquad \mathcal{H} \vdash \tau \leq \tau'}{\mathcal{H} \vdash m\langle\Theta\rangle(x_1 : \tau_1, \ldots) : \tau\ \texttt{extends}\ m\langle\Theta\rangle(x_1 : \tau_1', \ldots) : \tau'}$$

$$\frac{\mathcal{H} \vdash \tau_1' \blacktriangleleft \tau_1 \qquad \ldots \qquad \mathcal{H} \vdash \tau \leq \tau'}{\mathcal{H} \vdash m\langle\Theta\rangle(x_1 : \tau_1, \ldots) : \tau\ \texttt{implements}\ m\langle\Theta\rangle(x_1 : \tau_1', \ldots) : \tau'}$$

Figure 6.5: Hierarchy Typing, where $\overline{x}$ is a shorthand for $x, \ldots$ or $x; \ldots$

To prevent this mismatch between notions of precision, `Sub`'s return type should at least be a *substitutive* subtype ($<:$) of `Super`'s return type.

However, substitutive subtyping is also too lax for return types. While it address issues with typeability, it does not address issues with casts. Suppose that class $C\langle\rangle$ implements a method with return type **`dyn`**. Now suppose that the, as the code executes, an error is reported blaming the method implementation for failing to return a $\mathbb{Z}$. This would be perplexing but necessary if the class indicates that it inherits an interface with the same method but with the return type $\mathbb{Z}$. The same confusion would happen if a lambda closure were cast to an interface returning **`dyn`** but whose superinterface returns $\mathbb{Z}$. Inheritance provides a syntactic form of indirection, and to prevent such confusion it is important that return types in classes and interfaces always guarantee the contracts of return types in inherited interfaces are also satisfied. That is, return types must be covariant with respect to *pessimistic* subtyping ($\blacktriangleleft$) as well.

These two conditions are in fact necessary and sufficient for ensuring return types are well behaved both statically and dynamically. The notion of subtyping that not only implies but is even equivalent to the conjunction of substitutive and pessimistic is precisely inheritance ($\leq$), which we require of return types in our rules for method extension and implementation.

Consider next the input types. When the class or interface type of the receiver is known statically, the arguments are cast according to the input types of that *static* type's method. It should be the case that the more informative this type is, the more relaxed the casts can be. That is, the contracts of a static signature's input types should be guaranteed to be satisfied by the contracts of any inherited signature's input types. In

other words, input types must be contravariant with respect to *pessimistic* subtyping.

This condition is in fact sufficient for classes implementing interfaces, but not for interfaces extending interfaces. Suppose functional interface `Super`'s method accepts $\mathbb{Z}$ and subinterface `Sub` overrides that method to accept **`dyn`**. This more than broadens `Super`'s signature to accept more inputs, it also makes the signature more optimistic. When a lambda closure is cast to a functional interface, the signature of its implementation is *optimistically* compared to the interface's signature. So if a lambda closure's implementation were to accept an $I\langle\rangle$, this would be optimistically compatible with `Sub`'s signature. Since the cast to `Sub` succeeds, one would then expect a cast to `Super` to succeed. That is, one would expect weakening the type of a variable to not change the semantics of the program (provided it still type checks with the weaker type). But in this case the cast to `Super` would fail because the lambda closure's input type $I\langle\rangle$ is incompatible with `Super`'s input type $\mathbb{Z}$. In order to prevent this, it must be the case that every optimistic supertype of a subinterface's input type must also be an optimistic supertype of a superinterface's input type. Requiring input types of interfaces to be contravariant with respect to inheritance ($\leq$) ensures this behavior and implies contravariance with respect to pessimistic subtyping.

These restrictions might seem excessive. However, realize that the notion of subtyping in statically typed programming languages is precisely inheritance (restricted to concrete type). Thus our calculus is no more restrictive than statically typed languages. The restriction only limits how dynamism can vary across signatures. Interestingly, when we discuss our precision relation in the next section, we will see a similar need to restrict but in the

opposite direction: variations with respect to inheritance will be restricted rather than dynamism.

6.5 THE TRANSITION

Now that we understand the syntax and type system the programmer would be working within, we can finally discuss the primary goal of the chapter: guaranteeing a transition path from untyped structural code to typed nominal code. Our calculus and language are designed to enable programmers to replace a record (or lambda) with a (new) class and be guaranteed that this change will preserve the behavior of the program provided the class has sufficient structure to accommodate the various ways the record was being used. Similarly, programmers should be able to define interfaces describing how values are used and transparently replace dynamic types with those new interfaces provided those interfaces indeed sufficiently describe the ways the values were being used.

The gradual guarantee [Siek, Vitousek, Cimini, and Boyland, 2015] provides assurances on how a "correct" program *types* can be changed without changing the program's extensional compile-time and run-time behavior (i.e. the only changes have to do with type errors). This guarantee is defined in terms of a "precision" relation ($\sqsubseteq$) that extends the precision relation on types ($\precsim$) to the entire program so that type annotations can be changed. The guarantee ensures that replacing types with **`dyn`** preserves the typeability of the program and the semantics of the program except for possibly reducing occurrences of failed type casts. In other words, the

guarantee ensures that replacing **dyn**s with types preserves the typeability and semantics of the program except where the new types are incorrect.

Notice that the precision relation serves two roles: typing and semantics. In our setting, we need to separate these two roles. We want to provide semantic guarantees about changes to the program that do not necessarily preserve typeability and that are not necessarily restricted to just type annotations. In particular, we want to reason about programs with different nominal hierarchies, and consequently different grammars of valid types, so that we can provide guarantees about replacing records and lambdas with class instances and about inserting interfaces describing preexisting interactions.

For precision with respect to typing, we use the symbol $\sqsubseteq_{\preccurlyeq}$. For precision with respect to semantics, we use the symbol $\sqsubseteq_{\blacktriangleleft}$. As the notation suggests, the definitions of these relations are fairly similar. For many rules the only difference is that the former uses $\preccurlyeq$ whereas the latter uses $\blacktriangleleft$. As such, most rules are presented using a metavariable $\lll$ representing either $\preccurlyeq$ or $\blacktriangleleft$.

6.5.1 *Changing the Nominal Hierarchy*

Figure 6.6 provides the precision rules for the nominal hierarchy. Notice that the first rule indicates that we are only interested in well-formed hierarchies. In particular, the judgement $\vdash \mathcal{H} \sqsubseteq_{\preccurlyeq} \mathcal{H}'$ is *not* intended to guarantee that if $\mathcal{H}$ is well-formed then so is $\mathcal{H}'$. As we have already discussed, it is important for subinterfaces and superinterfaces to be consistent with the dynamism in their signatures, so the standard expectation

Hierarchy Precision $\vdash \mathcal{H} \sqsubseteq_{\lll} \mathcal{H} \qquad \mathcal{H} \sqsubseteq \mathcal{H} \vdash \mathcal{H} \sqsubseteq_{\lll} \mathcal{H}$

$$\frac{\vdash \mathcal{H} \qquad \vdash \mathcal{H}' \qquad \mathcal{H} \sqsubseteq \mathcal{H}' \vdash \mathcal{H} \sqsubseteq_{\lll} \mathcal{H}'}{\vdash \mathcal{H} \sqsubseteq_{\lll} \mathcal{H}'}$$

$$\frac{\mathcal{H} \sqsubseteq \mathcal{H}' \vdash \mathcal{H}_0 \sqsubseteq_{\lll} \mathcal{H}_0' \quad \mathcal{H} \sqsubseteq \mathcal{H}' \vdash \mathcal{M} \sqsubseteq_{\lll} \mathcal{M}'}{\mathcal{H} \sqsubseteq \mathcal{H}' \vdash \mathcal{H}_0; \mathcal{M} \sqsubseteq_{\lll} \mathcal{H}_0'; \mathcal{M}'} \qquad \frac{\mathcal{H} \sqsubseteq \mathcal{H}' \vdash \mathcal{H}_0 \sqsubseteq \mathcal{H}_0'}{\mathcal{H} \sqsubseteq \mathcal{H}' \vdash \mathcal{H}_0; \mathcal{M} \sqsubseteq_{\lll} \mathcal{H}_0'}$$

Material Precision $\mathcal{H} \sqsubseteq \mathcal{H} \vdash \mathcal{M} \sqsubseteq_{\lll} \mathcal{M}$

$$\frac{\begin{array}{c}\mathcal{I} = \texttt{interface}\ I\langle\Theta\rangle\ \texttt{extends}\ I_1\langle\sigma_1^1,...\rangle,...,I_1''\langle...\rangle,...\ \{m_1 s_1;...\} \\ \mathcal{I}' = \texttt{interface}\ I\langle\Theta\rangle\ \texttt{extends}\ I_1\langle\sigma_1^1,...\rangle,...\ \{m_1 s_1';...\} \\ I_1'' \notin \mathcal{H}' \quad ... \qquad \mathcal{H} \sqsubseteq \mathcal{H}' \vdash s_1 \sqsubseteq_{\lll}^{\preccurlyeq} s_1' \quad ...\end{array}}{\mathcal{H} \sqsubseteq \mathcal{H}' \vdash \mathcal{I} \sqsubseteq_{\lll} \mathcal{I}'}$$

$$\frac{\begin{array}{c}\mathcal{C} = \texttt{class}\ x : C\langle\Theta\rangle(\overline{f_1 : \tau_{f;1}};\Gamma)\ \texttt{implements}\ \overline{I_1\langle\overline{\sigma_1^1}\rangle, I_1''\langle...\rangle}\ \{\overline{m_1 s_1 \mapsto e_1}\} \\ \mathcal{C}' = \texttt{class}\ x : C\langle\Theta\rangle(\overline{f_1 : \tau_{f;1}'};\Gamma')\ \texttt{implements}\ I_1\langle\overline{\sigma_1^1}\rangle\ \{\overline{m_1 s_1' \mapsto e_1'}\} \\ \mathcal{H} \sqsubseteq \mathcal{H}' \vdash \Gamma \lll \Gamma' \qquad \mathcal{H} \sqsubseteq \mathcal{H}' \vdash \tau_1 \lll \tau_1' \quad ... \\ I_1'' \notin \mathcal{H}' \quad ... \qquad \mathcal{H} \sqsubseteq \mathcal{H}' \vdash s_1 \mapsto e_1 \sqsubseteq_{\lll}^{\lll} s_1' \mapsto e_1' \quad ...\end{array}}{\mathcal{H} \sqsubseteq \mathcal{H}' \vdash \mathcal{C} \sqsubseteq_{\lll} \mathcal{C}'}$$

Type Precision $\mathcal{H} \sqsubseteq \mathcal{H} \vdash \tau \lll \tau$

$$\frac{\mathcal{H} \mid \Theta \vdash \tau \quad \mathcal{H}' \mid \Theta \vdash \tau' \qquad \mathcal{H} \vdash \tau \blacktriangleleft \tau'}{\mathcal{H} \sqsubseteq \mathcal{H}' \vdash \tau \blacktriangleleft \tau'} \qquad \frac{\mathcal{H} \mid \Theta \vdash \tau \quad \mathcal{H}' \mid \Theta \vdash \tau' \qquad \tau \preccurlyeq \tau'}{\mathcal{H} \sqsubseteq \mathcal{H}' \vdash \tau \preccurlyeq \tau'}$$

Signature Precision $\mathcal{H} \sqsubseteq \mathcal{H} \vdash s \sqsubseteq_{\lll}^{\lll} s$

$$\frac{\mathcal{H} \sqsubseteq \mathcal{H}' \vdash \tau_1 \lll_i \tau_1' \quad ... \qquad \mathcal{H} \sqsubseteq \mathcal{H}' \vdash \tau \lll_o \tau'}{\mathcal{H} \sqsubseteq \mathcal{H}' \vdash \langle\Theta\rangle(x_1 : \tau_1,...) : \tau \sqsubseteq_{\lll_o}^{\lll_i} \langle\Theta\rangle(x_1 : \tau_1',...) : \tau'}$$

Implementation Precision $\mathcal{H} \sqsubseteq \mathcal{H} \vdash s \mapsto e \sqsubseteq_{\lll}^{\lll} s \mapsto e$

$$\frac{\mathcal{H} \sqsubseteq \mathcal{H}' \vdash s \sqsubseteq_{\lll_o}^{\lll} s' \qquad \mathcal{H} \sqsubseteq \mathcal{H}' \vdash e \sqsubseteq_{\lll} e'}{\mathcal{H} \sqsubseteq \mathcal{H}' \vdash s \mapsto e \sqsubseteq_{\lll_o}^{\lll} s' \mapsto e'}$$

Figure 6.6: Hierarchy Precision, where $\lll$ ranges over $\{\preccurlyeq, \blacktriangleleft\}$, and $\overline{x}$ is a shorthand for $x,...$ or $x;...$

of being able to replace arbitrary types in a signature with **dyn** will not result in a valid hierarchy. Instead, this judgement is intended to indicate that expressions typeable using $\mathcal{H}$ will also be typeable using $\mathcal{H}'$ *but* with one significant caveat.

Focus on the second rule for the judgement $\mathcal{H} \sqsubseteq \mathcal{H} \vdash \mathcal{H} \sqsubseteq_{\lll} \mathcal{H}$. This rule allows the more precise program to have classes and interfaces that are entirely missing from the less precise program (note that the notation is known to be confusing: the left-hand side is more precise and the right-hand side is less precise). Clearly removing classes and interfaces makes a program less typeable because it means even the types in the original program are no longer valid. Thus the caveat is that all types that become invalid by removing these classes and interfaces are also somehow removed from the program. For typing precision, this is done by replacing relevant types with **dyn**. For semantic precision, this can be done by replacing relevant typed with *either* **dyn** *or* some pessimistic supertype (since a removed class or interface might have implemented or extended an interface that was not removed).

To formalize this caveat, we use the judgement $\mathcal{H} \sqsubseteq \mathcal{H}' \vdash \tau \lll \tau'$. This judgement first checks that each type is valid according to its respective hierarchy (using an arbitrary kind context Θ in order to effectively ignore type variables). It then checks that the two types are related according to $\lll$ (the specific choice of hierarchy does not matter). Thus if τ is some $M\langle ... \rangle$ where M is not in $\mathcal{H}'$, then τ' must be **dyn** if $\lll$ is $\preccurlyeq$ or possibly some superinterface still in $\mathcal{H}'$ if $\lll$ is $\blacktriangleleft$.

Looking at the precision rule for classes, we can see that this judgement is used to check the precision of fields in the two class declarations. For precision typing, the use of $\preccurlyeq$ ensures both that any assignments

to fields in the former hierarchy will still type-check in the latter hierarchy and that any fetches of fields in the latter will still type-check in all contexts where fetches of fields in the former did (though possibly requiring dispatch modes to be recomputed). The former property holds because $\tau_e \triangleleft \tau \preccurlyeq \tau'$ implies $\tau_o \triangleleft \tau'$, and the latter property holds because $\preccurlyeq$ implies $<:$ (subsumptive subtyping).

6.5.2 *Changing Method Signatures*

Beyond changes in the existence of interfaces and classes and in field types, the methods of the nominal hierarchy can be changed as well. As with inheritance of methods, precision of methods needs to be done with care because of their dual role in typing and casts. For this reason, the precision judgements for signatures and signature implementations are parameterized by an (upper) relation to use for inputs (and implementations) and a (lower) relation to use for outputs. Notice that, unlike with inheritance, inputs are *covariant* with respect to this relation. This is because precision is *not* the same thing as overriding; it is changing the signature in the class or interface itself, not broadening it in some subclass or subinterface.

For class methods, the relation for input and output types is simply the relation for the notion of precision at hand, as one would expect. For interface methods, though, this relation is only used for output types, whereas the relation for input types is *always* $\preccurlyeq$ (which is the more restrictive of the two relations). This is because, for semantic precision, we have to ensure that any cast to this interface that would succeed in the more precise program would also succeed in the less precise program. In

particular, if we were to cast a lambda closure to this interface, we would check that the closure's signature is (optimistically) compatible with the interface's signature. This means we would check that the interface's input types are (optimistic) subtypes of the closure's input types. If the input type for the more precise interface were `Sub` and the closure's input type were also `Sub`, then this check would succeed. However, if the less precise interface's input type were `Super`, then this check would fail, thereby violating our desired gradual guarantee. As such, even for semantic precision we must use just $\preccurlyeq$ on input types of interface methods rather than the more permissive $\blacktriangleleft$. Note that this concern does not apply to classes since structural values cannot be cast to classes, and as such we can use $\blacktriangleleft$ for semantic precision of class-method input types.

6.5.3 *Changing Expressions*

Supposing we have a change in nominal hierarchies satisfying $\vdash \mathcal{H} \sqsubseteq_{\lll} \mathcal{H}'$, we now consider which corresponding changes to expressions we will provide guarantees for. Figure 6.7 provides a selection of the key precision rules for expressions.

6.5.3.1 *Changing Allocations*

The first two rules formally capture the main theoretical contribution of this chapter. One interpretation of them is that, once the appropriate structure of various values has been finalized, the programmer can write a class describing that structure and be guaranteed that allocations of those structures can be replaced with allocations of this class without changing

Near-Value $\quad \nu ::= x \mid v$

Expression Precision $\quad \mathcal{H} \sqsubseteq \mathcal{H} \vdash e \sqsubseteq_{\lll} e$

$$\frac{\begin{array}{c}\textbf{class}\ x : C\langle\overline{\alpha_1}\rangle(\overline{f_1 : \tau_{f;1}}; \overline{x_1 : \tau_{x;1}})\ \{\overline{m_1 s_1 \mapsto e_{m;1}}\} \in \mathcal{H} \\ \lll = \blacktriangleleft \implies C \notin \mathcal{H}' \qquad \mathcal{H} \sqsubseteq \mathcal{H}' \vdash e_{f;1} \sqsubseteq_{\lll} e'_{f;1} \quad ... \\ \mathcal{H} \sqsubseteq \mathcal{H}' \vdash s_1[\overline{\alpha_1 \mapsto \sigma_1}] \mapsto e_{m;1}[\overline{\alpha_1 \mapsto \sigma_1}, \overline{x_1 \mapsto \nu_1}] \sqsubseteq_{\blacktriangleleft}^{\lll} s'_1 \mapsto e'_{m;1} \quad ...\end{array}}{\mathcal{H} \sqsubseteq \mathcal{H}' \vdash \textbf{new}\ C\langle\overline{\sigma_1}\rangle(\overline{e_{f;1}}; \overline{\nu_1}) \sqsubseteq_{\lll} \textbf{new}\ (\overline{f_1 := e'_{f;1}})_x\{\overline{m_1 s'_1 \mapsto e'_{m;1}}\}}$$

$$\frac{\begin{array}{c}\textbf{class}\ x : C\langle\overline{\alpha_1}\rangle(\varnothing; \overline{x_1 : \tau_1})\ \{\lambda s \mapsto e\} \in \mathcal{H} \qquad \lll = \blacktriangleleft \implies C \notin \mathcal{H}' \\ \mathcal{H} \sqsubseteq \mathcal{H}' \vdash s[\overline{\alpha_1 \mapsto \sigma_1}] \mapsto e[\overline{\alpha_1 \mapsto \sigma_1}, \overline{x_1 \mapsto \nu_1}] \sqsubseteq_{\blacktriangleleft}^{\lll} s' \mapsto e'\end{array}}{\mathcal{H} \sqsubseteq \mathcal{H}' \vdash \textbf{new}\ C\langle\overline{\sigma_1}\rangle(\varnothing; \overline{\nu_1}) \sqsubseteq_{\lll} \textbf{new}\ \lambda s' \mapsto e'}$$

$$\frac{\mathcal{H} \sqsubseteq \mathcal{H}' \vdash s \mapsto e \sqsubseteq_{\preccurlyeq}^{\lll} s' \mapsto e'}{\mathcal{H} \sqsubseteq \mathcal{H}' \vdash \textbf{new}\ \lambda s \mapsto e \sqsubseteq_{\lll} \textbf{new}\ \lambda s' \mapsto e'}$$

$$\frac{\begin{array}{c}\mathcal{H} \sqsubseteq \mathcal{H}' \vdash e \sqsubseteq_{\lll} e' \\ \mathcal{H} \sqsubseteq \mathcal{H}' \vdash e_1 \sqsubseteq_{\lll} e'_1 \quad ... \qquad \mathcal{H} \sqsubseteq \mathcal{H}' \vdash \delta\langle n_\alpha\rangle(n_x) \sqsubseteq_{\lll} \delta'\langle n_\alpha\rangle(n_x)\end{array}}{\mathcal{H} \sqsubseteq \mathcal{H}' \vdash e\langle\sigma_1, ..., \sigma_{n_\alpha}\rangle(e_1, ..., e_{n_x})^{\delta} \sqsubseteq_{\lll} e'\langle\sigma_1, ..., \sigma_{n_\alpha}\rangle(e'_1, ..., e'_{n_x})^{\delta'}}$$

Dispatch Mode Precision $\quad \mathcal{H} \sqsubseteq \mathcal{H} \vdash \delta\langle n\rangle(n) \sqsubseteq_{\lll} \delta\langle n\rangle(n)$

$$\frac{}{\mathcal{H} \sqsubseteq \mathcal{H}' \vdash \delta.m\langle n_\alpha\rangle(n_x) \sqsubseteq_{\lll} \textbf{dyn}\langle n_\alpha\rangle(n_x)}$$

$$\frac{\begin{array}{c}\mathcal{H} \sqsubseteq \mathcal{H}' \vdash M\langle\Theta\rangle \blacktriangleleft M'\langle\sigma'_1, ...\rangle \qquad \mathcal{H} \vdash M\langle\Theta\rangle.ms \\ \mathcal{H}' \vdash M'\langle\sigma'_1, ...\rangle.ms' \qquad |ms| = \langle m, n_\alpha, n_x\rangle = |ms'| \qquad \mathcal{H} \sqsubseteq \mathcal{H}' \vdash s \sqsubseteq_{\blacktriangleleft}^{\blacktriangleleft} s'\end{array}}{\mathcal{H} \sqsubseteq \mathcal{H}' \vdash M.m\langle n_\alpha\rangle(n_x) \sqsubseteq_{\blacktriangleleft} M'.m\langle n_\alpha\rangle(n_x)}$$

$$\frac{M \in \mathcal{H} \qquad M \in \mathcal{H}'}{\mathcal{H} \sqsubseteq \mathcal{H}' \vdash M.m\langle n_\alpha\rangle(n_x) \sqsubseteq_{\preccurlyeq} M.m\langle n_\alpha\rangle(n_x)}$$

Figure 6.7: Expression Precision (selected rules), where $\lll$ ranges over $\{\preccurlyeq, \blacktriangleleft\}$, and $\overline{x}$ is a shorthand for $x, ...$ or $x; ...$

the semantics of the program (assuming the class was designed correction). Another interpretation is that removing a class from the hierarchy and replacing all of its corresponding instance allocations with an appropriate structural counterpart is guaranteed to preserve both the typeability and semantics of the program. Note that this guarantee applies *even if* the class implements some interfaces that are still be present in the program *even though* the structural counterpart does nothing to declare its intent to implement of those interfaces. Thus this mandates being able to dynamically impose (multiple) interfaces upon any structure that could possibly be converted into a class that could implement those interfaces. Given the importance of these rules, let us examine them in detail.

In the case of semantic precision, both rules explicitly require the class C to no longer be in the hierarchy $\mathcal{H}'$. This is necessary because structural values cannot be cast to class types. By requiring the class type to not in the less precise hierarchy, we ensure that this particular difference in casting behavior cannot be observed. This is the main reason why it is important to consider precision of expressions *with respect to* a change in hierarchies.

Both rules use "near-values" v. The method implementations in records and lambdas can reference local variables in the context. As execution proceeds, these local variables will eventually be substituted with values. A near-value essentially represents a local variable that may have or may have not yet been substituted for a value. What is important is that a near-value is either irreducible or will only ever be substituted with irreducible expressions. Thus it is fine for the class allocation to reference the near-value even if it never gets evaluated in the corresponding structural implementation. Note that it is also important that these near-

values are not assigned to fields; if they were then structural code would be able to access those fields in the class instance but the corresponding field accesses would fail on the structural value since the fields do not exist as they were not necessary to capture the local variables. (After all, lambda closures cannot even have fields.)

When comparing method signatures between class allocations and structural allocations, we can always use $\blacktriangleleft$ (the more permissive of the two relations) to compare return types. This is because the signatures of the structural values are not externally visible, so we only need to ensure that the class's implementation would still type-check with the structural value's signature. Pessimistic subtyping ensures this because $\tau_e \triangleleft \tau \blacktriangleleft \tau'$ implies $\tau_e \blacktriangleleft \tau'$, and in general our optimistic typing rules admit subsumption with respect to pessimistic subtyping.

6.5.3.2 *Changing Structural Methods*

The third rule for expression precision is shown to illustrate the precision requirements between method signatures of structures. Here we shown the rule for lambda expressions; the rule for records is similar, just more complex because records are syntactically more complex. The detail to note is that precision between return types is always restricted to $\preccurlyeq$, even in the case of semantic precision. The reason is that we need to ensure that any casts that could be successfully applied to the more precise lambda can also be applied to the less precise lambda. In particular, any optimistic supertype of the more precise lambda's return type must also be an optimistic supertype of the less precise return lambda's type. Precision ($\preccurlyeq$) ensures this property, whereas pessimistic subtyping ($\blacktriangleleft$) does not.

6.5.3.3 *Changing Dispatch Modes*

The final rule for expression precision is for invocation. This is mostly straightforward, but there are two details to note. One is that the type arguments must be exactly the same in both cases; they are concrete types after all, so there is no dynamism that can be changed anyways. The other is that the dispatch mode must be related.

When the less precise program uses dynamic dispatch, or when we are considering typing precision, this comparison of dispatch modes is uninteresting. However, there is an important subtlety to be aware of when considering semantic precision of statically resolved dispatch modes. Suppose the programmer determines that a variable of some interface type $I\langle\rangle$ only ever dynamically references instances of some class $C\langle\rangle$ implementing $I\langle\rangle$. They would like to know that refining the type of the variable to reflect this information (and get better performance) in fact preserves the semantics of the program. As such, we define semantic precision to permit more precise variables to be declared with pessimistic subtypes of their less precise counterparts.

This definition also captures the expectation that changing the type of a variable from some subinterface `Sub` to some superinterface `Super` would not change the (dynamic) semantics of the program (except for possibly permitting more successful executions). However, that expectation does not quite match reality. Recall that methods of subinterfaces and classes can have broader input types than methods of their superinterfaces. And recall that this means the types that (untyped) arguments are cast to depend on the (most precise) type of the receiver. As such, changing the program so that the receiver type changes from `Sub` to `Super` can in fact cause more

cast errors due to `Super` imposing more restrictive casts. So, rather than switch semantic precision entirely over to the more restrictive $\preccurlyeq$ relation to prevent this uncommon possibility, we instead check that the input types of the signature for the more precise dispatch mode are pessimistic subtypes than the input types of the less precise dispatch mode, thus ensuring that any casts imposed by the more precise dispatch mode are more restrictive than the casts for the less precise dispatch mode.

6.5.4 *The Static Gradual Guarantee*

At last we can formally state our first guarantee about transitioning between structural and nominal code with a property known as the static gradual guarantee.

Theorem 6.1. *For all nominal hierarchies $\mathcal{H}$ and $\mathcal{H}'$, expressions e and e', and types τ and τ',*

$$\vdash \mathcal{H} \sqsubseteq_{\preccurlyeq} \mathcal{H}' \quad \wedge \quad \mathcal{H} \sqsubseteq \mathcal{H}' \vdash e \sqsubseteq_{\preccurlyeq} e' \quad \wedge \quad \tau \preccurlyeq \tau' \quad \wedge \quad \mathcal{H} \mid \varnothing \mid \varnothing \vdash e \triangleleft \tau$$

$$\Downarrow$$

$$\exists e''.\ e' \equiv e'' \ (\mathsf{mod}\ \delta) \quad \wedge \quad \mathcal{H}' \mid \varnothing \mid \varnothing \vdash e'' \triangleleft \tau'$$

This theorem ensure that if a program is well-typed then any program that is less precise with respect to typing is necessarily also well-typed (modulo a possible need to recompute dispatch-mode annotations). In particular, according to our definition of typing precision, this implies that a program is well-typed if it is possible to add classes and interfaces, replace all records and lambdas with allocations of classes, and replace all occurrences of **`dyn`** with concrete types such that the resulting program is

well-typed. In other words, any program with a viable path towards being statically well-typed is guaranteed to be gradually well-typed, even if that path requires changing structural code to nominal code.

Of course, being well-typed in a gradually typed language guarantees very little about actual execution behavior. Thus we need to provide another guarantee that the existence of this path towards a fully statically typed program that necessarily never gets stuck also ensures the current program never gets stuck. That is, we need a corresponding guarantee about the semantics of the calculus.

6.6 SEMANTICS

The semantics of our calculus was carefully designed to simultaneously facilitate the proof of our transition guarantee and to enable an efficient implementation of our language. Figure 6.8 provides rules specifically selected to illustrate key properties of our semantics.

6.6.1 *Semantic Expressions, Values, and Heaps*

Now we discuss aspects of the calculus that are specific to the semantics and not visible at the surface level. Expressions include locations, i.e. references into a mutable heap, and explicit casts, which are only used to cast values to be returned by methods (more on this later). Values are either booleans, integers, reals, or references into the heap. Evaluation contexts are as one would expect except that dispatch modes of the form $M.f$ prevent the corresponding $e.f$ from being evaluated as a field access.

$$
\begin{array}{lll}
\text{Value} & v & ::= b \mid i \mid r \mid \ell \\
\text{Evaluation} & E & ::= \odot \mid \texttt{let}\ x : \tau := E\ \texttt{in}\ e \mid \texttt{if}\ E\ \texttt{then}\ e\ \texttt{else}\ e \\
\text{Context} & & \mid E + e \mid v + e \mid \texttt{new}\ C\langle\sigma, ...\rangle(e, ...; v, ..., E, e, ...) \\
& & \mid \texttt{new}\ C\langle\sigma, ...\rangle(v, ..., E, e, ...; v, ...) \\
& & \mid \texttt{new}\ (f := v; ...; f := E; f := e; ...)_x\{ms \mapsto e; ...\} \\
& & \mid E.f \mid E.f := e \mid v.f := E \mid E \approx e \mid v \approx E \\
& & \mid E.f\langle\sigma, ...\rangle(e, ...)^{M.f} \mid v.f\langle\sigma, ...\rangle(v, ..., E, e, ...)^{M.f} \\
& & \mid E\langle\sigma, ...\rangle(e, ...)^{M.\lambda} \mid v\langle\sigma, ...\rangle(v, ..., E, e, ...)^{M.\lambda} \\
& & \mid E\langle\sigma, ...\rangle(e, ...)^{\texttt{dyn}} \mid v\langle\sigma, ...\rangle(v, ..., E, e, ...)^{\texttt{dyn}} \\
& & \mid \texttt{cast}\ E\ \texttt{to}\ \tau \mid \texttt{cast}\ E\ \texttt{to}\ \tau\ \texttt{for}\ \ell.m\langle\sigma, ...\rangle(\nu, ...) \\
\hline
\text{Location} & \ell & \\
\text{Mutability} & \mu & ::= \mathsf{ro} \mid \mathsf{rw} \\
\text{Imposition} & \iota & ::= I\langle\tau, ...\rangle \\
\text{Heap Value} & h & ::= C\langle\sigma, ...\rangle(v, ...; v, ...) \mid (f \mapsto_\mu v; ...)_x\{ms \mapsto e; ...\}_{\iota,...} \\
& & \mid \lambda_{\iota,...} s \mapsto e \\
\text{Heap} & H & ::= \ell \mapsto h; ...
\end{array}
$$

Transition Relation $\mathcal{H} \vdash H \mid e \to_\ell H \mid e$

$$\frac{\mathcal{H} \mid H \to H' \vdash v : \tau}{\mathcal{H} \vdash H \mid E[\texttt{let}\ x : \tau := v\ \texttt{in}\ e] \to_\ell H' \mid E[e[x \mapsto v]]}$$

$$\frac{\ell \notin H}{\mathcal{H} \vdash H \mid E[\texttt{new}\ \lambda s \mapsto e] \to_\ell H; \ell \mapsto \lambda s \mapsto e \mid E[\ell]}$$

$$\frac{\mathcal{M} \mid H \to H' \vdash \ell'.\lambda\langle\sigma_1, ...\rangle(v_1, ...)^\delta \to e}{\mathcal{H} \vdash H \mid E[\ell'\langle\sigma_1, ...\rangle(v_1, ...)^\delta] \to_\ell H' \mid E[e]}$$

$$\frac{\mathcal{M} \mid H \to H' \vdash \ell'.f\langle\sigma_1, ...\rangle(v_1, ...)^\delta \to e}{\mathcal{H} \vdash H \mid E[\ell'.f\langle\sigma_1, ...\rangle(v_1, ...)^\delta] \to_\ell H' \mid E[e]}$$

$$\frac{\mathcal{H} \mid H \to H' \vdash v : \tau}{\mathcal{H} \vdash H \mid E[\texttt{cast}\ v\ \texttt{to}\ \tau] \to_\ell H' \mid E[v]}$$

$$\frac{\begin{array}{c}\mathcal{H} \mid H \to H_0 \vdash v : \tau \\ H_0 \vdash \ell' \mapsto \iota_1, ... \qquad \mathcal{H} \mid H_0 \to H_1 \vdash v : \iota_1.m\langle\sigma_1, ...\rangle(v_1, ...) \quad ... \\ \mathcal{H} \mid H_{n-1} \to H_n \vdash v : \iota_n.m\langle\sigma_1, ...\rangle(v_1, ...)\end{array}}{\mathcal{H} \vdash H \mid E[\texttt{cast}\ v\ \texttt{to}\ \tau\ \texttt{for}\ \ell'.m\langle\sigma_1, ...\rangle(v_1, ...)] \to_\ell H_n \mid E[v]}$$

Figure 6.8: Semantics (selected rules)

The heap is a (partial) mapping of locations to heap values, which are class instances, records, or lambda closures. Importantly, records and lambda closures have a list of interfaces that are imposed upon them. These lists are updated via monotonic casts, which are discussed in more detail next.

6.6.2 *Implicit Casts*

The semantics for a gradually typed language are usually given by a type-directed translation to a cast language, a process known as cast insertion. This is useful for showing that the cast-insertion implementation of the language is type-safe, i.e. programs only get stuck due to casts. Unfortunately our language is not implemented through cast insertion; although cast insertion is a part of the implementation, it is only a small part. In particular, invocation is implemented instead through v-tables, interface tables, and dictionary lookups. This implementation does rely on low-level invariants that casts help maintain, say by dynamically generating interface tables for structural values, but the calculus that properly formalizes our implementation so that we could formally prove soundness would unfortunately distract from the more important contributions of this work.

As such, our semantics instead asserts that casts are implicitly everywhere. This then guarantees that variables are only ever substituted with appropriate values, fields only ever contain appropriate values, and methods only ever return appropriate values. As such, many implicit casts can be proven to be unnecessary, and so our implementation instead works by removing unnecessary casts.

This notion of implicit casts can be seen in the rule for `let` expressions. This rule first checks that the value has the necessary type, possibly updating the heap to monotonically cast the value to make it so, and only if that is successful does it substitute the value into the body of the `let` expression. Thus, this substitution is only ever done if the variable was assigned an appropriate value.

The semantics of our monotonic casts [Siek, Vitousek, Cimini, Tobin-Hochstadt, et al., 2015; Vitousek, Kent, et al., 2014; Vitousek, Swords, and Siek, 2017] are shown in Figure 6.9. No check is done if the target type is **`dyn`**. Otherwise, primitive values trivially have the appropriate target type. References to class instances can be cast (without change) to any pessimistic supertype of the instantiated class. References to heap values with impositions (given by the judgement $\mathcal{H} \vdash \ell \mapsto \iota, ...$) can be cast (without change) to any pessimistic subtype of any of the impositions. Alternatively, references to records can be cast to any interface type that is compatible with its existing impositions. (The judgement $\mathcal{H} \vdash \iota_1, ... : I\langle\tau, ...\rangle$ indicates that $I\langle\tau, ...\rangle$ is the most precise instantiation of I that every imposition in $\iota_1, ...$ is a pessimistic subtype of if it inherits I at all.) Notice that with records nothing is done to check that the record has the appropriate method—this check is only done when the method is actually invoked, permitting records to provide partial implementations and preventing cast performance from being slowed down by many (potentially unnecessary) checks. On the other hand, casts of lambda closures immediately check that the target interface is a functional interface whose sole signature is optimistically compatible with the closure's. This enables us to build an optimized interface table and furthermore, if the signature happen

Let Context $L ::= \odot \mid \textbf{let}\ x : \tau := x\ \textbf{in}\ L$

Run Time Value Casting $\mathcal{H} \mid H \to H \vdash v : \tau$

$$\frac{}{\mathcal{H} \mid H \to H \vdash v : \textbf{dyn}} \qquad \frac{}{\mathcal{H} \mid H \to H \vdash b : \mathbb{B}}$$

$$\frac{}{\mathcal{H} \mid H \to H \vdash i : \mathbb{Z}} \qquad \frac{}{\mathcal{H} \mid H \to H \vdash r : \mathbb{R}}$$

$$\frac{\ell \mapsto C\langle\sigma_1, ...\rangle(...;...) \in H \quad \mathcal{H} \vdash C\langle\sigma_1, ...\rangle \blacktriangleleft \tau}{\mathcal{H} \mid H \to H \vdash \ell : \tau} \qquad \frac{H \vdash \ell \mapsto ..., \iota, ... \qquad \mathcal{H} \vdash \iota \blacktriangleleft \tau}{\mathcal{H} \mid H \to H \vdash \ell : \tau}$$

$$\frac{\ell \mapsto (f_1 \mapsto_{\mu_1} v_1; ...)_x\{m_1 s_1 \mapsto e_1; ...\}_{\iota_1,...} \in H \quad \forall I.\exists \tau_1, ... \mathcal{H} \vdash \iota_1, ..., \iota : I\langle\tau_1, ...\rangle}{\mathcal{H} \mid H \to H[\ell \mapsto (f_1 \mapsto_{\mu_1} v_1; ...)_x\{m_1 s_1 \mapsto e_1; ...\}_{\iota_1,..,\iota}] \vdash \ell : \iota}$$

$$\frac{\ell \mapsto \lambda_{\iota_1,...} s \mapsto e \in H \qquad \forall I'.\exists \tau'_1, ... \mathcal{H} \vdash \iota_1, ..., I\langle\tau_1, ...\rangle : I'\langle\tau'_1, ...\rangle \quad \textbf{interface}\ I\langle\alpha_1, ...\rangle\ \{\lambda s'\} \in \mathcal{H} \qquad \mathcal{H} \vdash s \triangleleft s'[\alpha_1 \mapsto \tau_1, ...]}{\mathcal{H} \mid H \to H[\ell \mapsto \lambda_{\iota_1,...,I\langle\tau_1,...\rangle} s \mapsto e] \vdash \ell : I\langle\tau_1, ...\rangle}$$

Invocation Semantics $\mathcal{H} \mid H \to H \vdash \ell.m\langle\sigma, ...\rangle(v, ...)^{\delta} \to e$

$$\frac{\begin{array}{c}\mathcal{H} \vdash M\langle\alpha_1, ...\rangle.m\langle\alpha'_1, ...\rangle(x_1 : \tau'_1, ..., x_n : \tau'_n) : \tau \qquad \mathcal{H} \mid H_0 \vdash \ell \leq M\langle\tau_1, ...\rangle \\ \mathcal{H} \mid H_0 \to H_1 \vdash v_1 : \tau'_1[\alpha_1 \mapsto \tau_1, ..., \alpha'_1 \mapsto \sigma_1, ...] \quad ... \\ \mathcal{H} \mid H_{n-1} \to H_n \vdash v_n : \tau'_n[\alpha_1 \mapsto \tau_1, ..., \alpha'_1 \mapsto \sigma_1, ...] \\ H_n \to H' \vdash \ell.m\langle\alpha'_1, ...\rangle(x_1 : \tau'_1, ..., x_n : \tau'_n)^{M.m} \mapsto e \\ e' = e[\alpha'_1 \mapsto \sigma_1, ..., x_1 \mapsto v_1, ..., x_n \mapsto v_n]\end{array}}{\mathcal{H} \vdash H_0 \to H' \vdash \ell.m\langle\sigma_1, ...\rangle(v_1, ..., v_n)^{M.m} \to e'}$$

$$\frac{\begin{array}{c}H \to H' \vdash \ell.m\langle\alpha'_1, ...\rangle(x_1 : \tau'_1, ..., x_n : \tau'_n)^{\textbf{dyn}} \mapsto e \\ e' = e[\alpha'_1 \mapsto \sigma_1, ..., x_1 \mapsto v_1, ..., x_n \mapsto v_n]\end{array}}{\mathcal{H} \vdash H \to H' \vdash \ell.m\langle\sigma_1, ...\rangle(v_1, ..., v_n)^{\textbf{dyn}} \to e'}$$

Figure 6.9: Cast and Invocation Semantics

Run Time Method Lookup $H \to H \vdash \ell.m\langle\Theta\rangle(\Gamma)^{\delta} \mapsto e$

$$\frac{\ell \mapsto C\langle\sigma_1,...\rangle(...;v_1,...) \in H \quad \textbf{class}\ x : C\langle\alpha_1,...\rangle(...;x_1:\tau_1,...)\ \{...;m\langle\Theta\rangle(\Gamma):\tau \mapsto e;...\} \in \mathcal{H} \quad e' = \textbf{cast}\ e[\alpha_1 \mapsto \sigma_1,...,x \mapsto \ell, x_1 \mapsto v_1,...]\ \textbf{to}\ \tau[\alpha_1 \mapsto \sigma_1,...]}{H \to H \vdash \ell.m\langle\Theta\rangle(\Gamma[\alpha_1 \mapsto \sigma_1,...])^{M.m} \mapsto e'}$$

$$\frac{\ell \mapsto C\langle...\rangle(...;...) \in H \qquad \mathcal{H} \mid H \vdash \ell.m\langle\Theta\rangle(x_1:\tau_1,...)^{C.m} \mapsto e}{H \to H \vdash \ell.m\langle\Theta\rangle(x_1:\tau_1,...)^{\textbf{dyn}} \mapsto \textbf{let}\ x_1:\tau_1 := x_1\ \textbf{in}\ ...\ e}$$

$$\frac{\ell \mapsto (...)_x\{...;m\langle\Theta\rangle(x_1:\tau_1,...):\tau \mapsto e;...\}_{...} \in H \quad e' = \textbf{let}\ x_1:\tau_1 := x_1\ \textbf{in}\ ...\ \textbf{cast}\ e[x \mapsto \ell]\ \textbf{to}\ \tau}{H \to H \vdash \ell.m\langle\Theta\rangle(x_1:\tau_1,...)^{\textbf{dyn}} \mapsto e'}$$

$$\frac{\ell \mapsto \lambda_{...}\langle\Theta\rangle(x_1:\tau_1,...):\tau \mapsto e \in H}{H \to H \vdash \ell.\lambda\langle\Theta\rangle(x_1:\tau_1,...)^{\textbf{dyn}}:\tau \mapsto \textbf{let}\ x_1:\tau_1 := x_1\ \textbf{in}\ ...\ \textbf{cast}\ e\ \textbf{to}\ \tau}$$

$$\frac{H \vdash \ell \mapsto \iota,... \qquad H \vdash \ell.m\langle\Theta\rangle(x_1:\tau_1,...)^{\textbf{dyn}} \mapsto L[\textbf{cast}\ e\ \textbf{to}\ \tau]}{H \to H \vdash \ell.m\langle\Theta\rangle(x_1:\tau_1,...)^{M.m} \mapsto L[\textbf{cast}\ e\ \textbf{to}\ \tau\ \textbf{for}\ \ell.m\langle\Theta\rangle(x_1,...)]}$$

$$\frac{\ell \mapsto (...,f \mapsto_{\mu} \ell',...)_x\{m_1s_1 \mapsto e_1;...\}_{\iota_1,...} \quad H \vdash \ell'.\lambda\langle\Theta\rangle(x_1:\tau_1,...)^{\textbf{dyn}} \mapsto L[\textbf{cast}\ e\ \textbf{to}\ \tau] \quad H' = H[\ell \mapsto (...,f \mapsto_{\mathsf{ro}} \ell',...)_x\{m_1s_1 \mapsto e_1;...\}_{\iota_1,...}]}{H \to H' \vdash \ell.f\langle\Theta\rangle(x_1:\tau_1,...)^{M.f} \mapsto L[\textbf{cast}\ e\ \textbf{to}\ \tau\ \textbf{for}\ \ell.f\langle\Theta\rangle(x_1,...)]}$$

Figure 6.9 (contd.): Cast and Invocation Semantics

to be pessimistically compatible, this table can point to a variant of the implementation that skips the argument casts entirely.

6.6.3 *Allocation*

The expressions allocating values onto the heap proceed as one would expect. We provide the rule for lambdas to illustrate one formal device we use. Note that the reduction judgement is annotated with a location. This

annotation is used by the allocation rules to indicate which location was allocated. This is the only source of non-determinism in the calculus, so by making it explicit we can compare reductions of programs that make the same non-deterministic choices.

6.6.4 *Invocation*

The semantics of invocations use a judgement defined in Figure 6.9. If the dispatch mode is some class or interface $M.m$, this judgement first casts the arguments to the parameter types expected by M, using the receiver's run-time instantiation of M and the invocation's type arguments to determine the types of any type variables in the parameter types. Note that these casts can normally be fully inserted at the call site at compile time; the only exception is when **`dyn`** occurs in the type arguments of the receiver's static type, in which case the type arguments to use need to be fetched from the receiver at run time. After these casts, the judgement uses another judgement to determine the receiver's run-time implementation of a method according to its signature *and* its dispatch mode. Note that the dispatch mode is only used here to distinguish static dispatch from dynamic dispatch—all static dispatch modes have the same implementation. This means our implementation can associate with each heap value two implementations of a method: the one that is retrieved through the v-table or interface table and is optimized for static dispatch, and the one that is retrieved through a dictionary and is optimized for dynamic dispatch.

If the receiver is a class and the dispatch mode is static, the implementation simply passes the (guaranteed to be valid) arguments straight through to the method implementation, but it does at least make sure to cast its own returned value to its own return type (in case the method implementation only optimistically type-checked). If the dispatch mode is instead dynamic, then the implementation casts the arguments to the method implementation's input types and then defers to the static implementation.

If the receiver is a structural value and the dispatch mode is dynamic, then the implementation first casts the arguments to the method implementation's input types, then supplies these (now valid) arguments to the method implementation, and finally casts its own returned value to its own return type. If the dispatch mode is instead static, then the implementation defers to the dynamic implementation (if it exists) but refines the cast of the returned value to also cast the return type to any return types expected by any of the interfaces that have been imposed upon the receiver. Lastly, as a special case to support more structural patterns, if the receiver is a record and the dispatch mode is static but the implementation is provided by a field of the receiver rather than the method, then the appropriate implementation is looked up from the value of that field, its return cast changed to check the impositions on the receiver, and the field is made immutable so that this implementation can be stored directly into the interface table for faster lookups in the future.

6.6.5 *The Dynamic Gradual Guarantee*

At last we can formally state our second guarantee about transitioning between structural and nominal code with a property known as the dynamic gradual guarantee, which is implied by the following two theorems (which reference the straightforward extension of semantic precision to heaps).

Theorem 6.2. *For all nominal hierarchies $\mathcal{H}$ and $\mathcal{H}'$, heaps H_1, H_1', and H_2, expressions e_1, e_1', and e_2, and locations ℓ,*

$$\vdash \mathcal{H} \sqsubseteq_{\blacktriangleleft} \mathcal{H}' \quad \wedge \quad \mathcal{H} \sqsubseteq \mathcal{H}' \vdash H_1 \mid e_1 \sqsubseteq_{\blacktriangleleft} H_1' \mid e_1' \quad \wedge \quad \mathcal{H} \vdash H_1 \mid e_1 \to_\ell H_2 \mid e_2$$

$$\Downarrow$$

$$\exists H_2', e_2'.\ \mathcal{H} \vdash H_1' \mid e_1' \to_\ell H_2' \mid e_2'$$

Theorem 6.3. *For all nominal hierarchies $\mathcal{H}$ and $\mathcal{H}'$, heaps H_1, H_1', H_2, and H_2', expressions e_1, e_1', e_2, and e_2', and locations ℓ,*

$$\bigwedge \begin{array}{c} \vdash \mathcal{H} \sqsubseteq_{\blacktriangleleft} \mathcal{H}' \wedge \mathcal{H} \sqsubseteq \mathcal{H}' \vdash H_1 \mid e_1 \sqsubseteq_{\blacktriangleleft} H_1' \mid e_1' \\ \mathcal{H} \vdash H_1 \mid e_1 \to_\ell H_2 \mid e_2 \wedge \mathcal{H} \vdash H_1' \mid e_1' \to_\ell H_2' \mid e_2' \end{array}$$

$$\Downarrow$$

$$\exists ..., H_n', e_n'. \ ... \wedge \mathcal{H} \vdash H_{n-1}' \mid e_{n-1}' \to_\ell H_n' \mid e_n' \wedge \mathcal{H} \sqsubseteq \mathcal{H}' \vdash H_2 \mid e_2 \sqsubseteq_{\blacktriangleleft} H_n' \mid e_n'$$

These theorems ensure that reduction (eventually) preserves semantic precision and that less precise programs always make progress when more precise programs do. In particular, according to our definition of typing precision, this implies that a program does not get stuck if it is possible to add classes and interfaces, replace all records and lambdas with allocations

of classes, and replace all occurrences of **`dyn`** with concrete types such that the resulting program does not get stuck. In other words, in combination with the static gradual guarantee, any program with a viable path towards being statically well-typed (and necessarily not getting stuck) is guaranteed to be gradually well-typed and to not get stuck, even if that path requires changing structural code to nominal code. Unlike Thorn [Wrigstad et al., 2010], this even includes situations in which structural values need to appear like nominal values implementing nominal interfaces.

Thus, our calculus and implementation provide a language in which structural values are strictly more expressive than their nominal counterparts, including being castable to nominal interfaces, with the one exception that they cannot be cast to nominal classes. Furthermore, transitioning structural components of the program to (correctly) use nominal patterns is guaranteed to preserve the semantics of the program. Of course, this is great in theory, but in practice gradual typing has a history of significant issues with large overheads caused by the casts ensuring safety [Takikawa et al., 2016]. Next we demonstrate that the design of our calculus enables an implementation that exhibits little-to-no overhead relative to the performance of fully untyped programs.

6.7 IMPLEMENTATION

Our implementation, called *MonNom*, takes careful detail to make common cases fast and to ensure low-level efficiency techniques wherever possible. However, because the language is new and still quite small and therefore no realistic programs in it exist, we are restricted to evaluat-

ing microbenchmarks. Therefore, we do *not* employ several optimization techniques that are used in practice but may threaten the validity of our results. These are techniques such as inference, caching, and speculation, which are quite likely to be far more effective for small languages with simplistic benchmarks than they will be for large languages and realistic benchmarks. There is one exception to this, which we believe to be justified as explained below in Section 6.7.5.

The MonNom runtime is implemented using LLVM's [Lattner and Adve, 2004; The LLVM Project, 2019] JIT libraries, although currently all programs are fully compiled at the start of the runtime executable. We again use the Boehm-Demers-Weiser conservative garbage collector to manage our heap-allocated values [Demers et al., 1990]. The rest of this section explains the major aspects of our particular implementation techniques.

6.7.1 *Primitives*

We represent primitive values differently depending on whether they are statically or dynamically typed, a technique we already employed in Nom, albeit slightly improved here, as explained below. In particular, besides reference values to heap-allocated objects, there are three kinds of primitive values: Booleans, 64-bit Integers, and 64-bit IEEE double-precision floating point numbers. All of these fit into 64 bits. In typed code, primitive types are used in their raw representation, but in untyped code, they need to be "packed" to also carry type information. Since pointers are 64-bit aligned, the lower three bits can carry type information, of which we use two for

packing – this is a common technique in dynamically typed languages. This would usually leave us with 62 bits to encode integers and floating point numbers; which makes integers less useful to express things that rely on (multiples of) 64-bit integers, such as bitmaps, encryption algorithms, IP addresses, and so on, and floating point numbers would lose a lot of precision and potentially special values such as NaN and infinity.

Nom and Grift [Kuhlenschmidt, Almahallawi, and Siek, 2019] use the 62-bit encoding; Grift allocates all floating point numbers on the heap, whereas Nom just accepts the loss of precision, following Self [Chambers, Ungar, and Lee, 1989].

Instead, Nom allocates certain packed values on the heap (those are called "boxed"), and leaves others on the stack. Recall that we use the lower two bits of a 64-bit value as a flag for which kind of value it is. Reference values use "00" as their flag and are thus unchanged. Integers use "11", and floats use both "01" and "10". Booleans that are being packed are converted into a reference to one of two preallocated Boolean object instances. When an integer is being packed, we first do a *funnel shift* by three bits to the left; that is, all bits are moved three positions to the left, and the formerly three most-significant bits are now the three least-significant bits, but in the same order as before. That is, the sign bit of the integer is now in the third-least-significant position. If the other two least-significant bits are the same as the sign-bit, the number is closer to zero than any other possible combination, hence it is a more common integer value, and those are the values we leave on the stack. All other integers are allocated as reference values on the heap. All in all, this means that integers between -2^{61} and $2^{61} - 1$ are always on the stack, either packed or unpacked, and all other integers between -2^{63} and $2^{63} - 1$ are boxed into reference values when packed.

When a float is packed, we again do a funnel shift by three bits to the left; again the sign bit is the third-least significant bit. If the two least-significant bits are either "01" or "10", the value can be left as-is, with one

exception: the floating point value 0.0 is expressed as the 64-bit value 1; conversely if every bit except the two least significant bits "01" (after the funnel shift) is 0, that value has to be allocated on the heap, too, because it collides with the encoding for 0.0. Many relatively short floating point numbers are using the "10" flag, such as all the lower "round" numbers, so a reasonable number of floats will not be allocated on the heap when packed.

Unpacking/unboxing are the straightforward reversals of the above operations.

6.7.2 *Interlude: Monotonic Casting to Generic Interfaces*

To support run-time type checks, gradual typing requires generics to be *reified*, that is, instances of a generic type like `List`⟨*String*⟩ need to be tagged such that it is clear that they are instances of class `List` with type argument `String`. This is usually done per instance, but requires that there is additional space in each instance per type argument instead of just a fixed-size tag for a single nominal class. For class instances created using their explicit constructors, that is easy, as the space can be allocated along with the rest of the object. However, for records and lambdas, it is unclear at the type of their allocation which type they will be cast to, and hence how many slots for type arguments there need to be. Our solution is to keep track of the maximum number of type arguments any type a record or lambda has been cast to for each record/lambda allocation site. New instances of that record/lambda are allocated with enough space to accomodate that number of type arguments when they encounter such a

cast. Any record or lambda that does not have enough space for the type arguments of the type it is cast to needs to create a special copy of that type's descriptor that includes the type arguments there – this is expensive, usually only a few instances will need to do that, as newer instances will already have enough space. There is yet another, even better optimization that covers what we believe to be a common case, explained below in Section 6.7.5.

6.7.3 *Heap Values*

Given a reference to a heap value, the reference points to a 64-bit block. If the heap value is a record or a lambda closure that has not yet been cast to any interface, then the lower two bits will indicate whether it is a record or lambda closure and the upper 32 bits denote how many type-argument slots have been preallocated elsewhere in the memory block, as discussed in Section 6.7.2. If the heap value is a class instance or a record or lambda closure that has been cast, then the lower two bits will be clear and the 64 bits will be a pointer to a virtual method table (*v-table*, see Section 6.7.4), which has the necessary information to determine whether the heap value is a class instance or a record or a lambda closure, but which only provides a lower bound for the number of type arguments that were preallocated.

For a class instance, the referenced memory block extends below the reference to include fields at statically determined offsets, and up to hold any type argument references, if the class is generic.

For a record, the referenced memory block extends below the reference to include a pointer to an immutable hidden-class dictionary (shared

across all records allocated by the same code site), a pointer to a mutable field dictionary (specific to the record), the values of the local variables in the closure, and the variables used in the constructor. Dictionary entries that are monotonically locked down (i.e. changed to read-only) at run-time are marked with a bit set on third-least significant bit (we exploit the fact that integers and floats are never locked in this fashion). Preallocated slots for type arguments, if there are any for the record, are found above the reference. The hidden-class dictionary is a hashmap from field/method names to field offsets or method-overloading descriptors (tagged accordingly). A method-overloading descriptor is an immutable association list of arities associated with code pointers to that overloadings implementation.

For a lambda closure, the referenced memory block extends below the reference to include a pointer to a lambda descriptor (shared across all lambda closures allocated by the same code site), then the values of local variables in the closure. Preallocated slots for type arguments, if there are any for the record, are found above the reference. The lambda descriptor provides the code pointer for the lambda implementation as well as a description of the signature to be used in run-time casts.

6.7.4 *V-Tables*

The lowest bit of the v-table pointer indicates whether the remaining bits point to a virtual-method table with an immutable association-list interface table or to a linked-list interface table. The former is used for interface tables whose layout could be generated at compile time, which is all interface tables except for interface tables for record or lambda closures

that have been cast to multiple unrelated interfaces. The association list specifies for each implemented interface the offset within the v-table at which its method table can be found. The linked list specifies the type arguments and method table for each interface. Initially the method table indicates to use a standard dictionary-lookup process, but this linked list is specific to the heap value and so the method tables are updated with the result of the lookup in order to make subsequent dispatches quick.

For class instances, the virtual-method table is very similar to Nom's (Section 5.8.1), except that there is a precompiled function rather than an association list for dynamic dispatcher lookup. That is, on a dynamic method call on the class, the precompiled dispatcher lookup function is accessed through the method table. It takes the "name" of the method (a unique number for each possible method name string), the number of type arguments, and the number of value arguments, and returns a dispatcher function. This dispatcher function takes the right number of type and value arguments and does the appropriate bounds and type checks, if any. While not currently a feature of MonNom, the original use of the dispatchers was also to support overloading, by using the type checks to implement a decision tree of which method to dispatch to, if any.

To support monotonically casting records as described in Section 6.7.2, there are up to two virtual-method-table layouts per interface declaration in the program. Every generic interface declaration has a virtual-method-table layout for records with an insufficient number of slots for preallocated type arguments. This layout includes room in the table to store the type arguments. As such, a new virtual-method table with this layout is allocated each time such a record is cast. In order to reduce such dynamic allocations, every generic interface declaration also has an immutable

virtual-method table for records with sufficient room for type arguments. In this case the type arguments are stored in the record and so the same such virtual-method table can be used by all such records, avoiding the need for dynamic allocation.

To support monotonically casting lambda closures as described in Section 6.7.2, there are four virtual-method-table layouts per functional interface declaration (i.e. interfaces that only declare a λ-method) in the program due to two degrees of freedom. One degree of freedom is the same as with records: whether type arguments are stored in the virtual-method table or in the lambda closure. The other degree of freedom is specific to lambda closures: whether the implementation's signature pessimistically or just optimistically satisfies the functional interface's signature. This is determined immediately upon (eagerly) casting the lambda closure. If the lambda closure's signature pessimistically satisfies the functional interface's signature, then the virtual-method table is optimized to bypass casts to the implementation's parameter types and from the implementation's return type.

6.7.5 *The Single-Target-Type Hypothesis*

For our intended applications of gradual typing, we suspect that most structural values allocated by a given code site will only ever be cast to one *instantiated* interface type with no reference to type variables in scope. Even better, the cast site will usually be the exact same for all those values. For example, lambda expressions are very often written directly as arguments to the typed method whose parameter they will be cast to.

As such, we associate an initially null virtual-method-table pointer and an initially null cast-site identifier with every structural allocation site.

When a not-yet-cast structural value is cast to an interface type with no reference to type parameters in the context, the cast-site identifier associated with the value's allocation site is checked. If it matches the current cast-site, then the associated virtual-method-table pointer is simply copied into the structural value. If not but the associated virtual-method-table pointer is not null, then it is checked whether it was generated for the interface type at hand, in which case it is copied into the structural value. If the virtual-method table is not a match for the interface type at hand, then the pointer is replaced with null and the cast-site identifier is replaced with a garbage identifier so that no such checks are done again. That is, this is *not* a cache, just an optimization that saves significant time when applicable and otherwise is nearly no overhead (just a dereference and equality check).

6.8 EVALUATION

MonNom is even more unique in its approach to gradual typing than Nom was. To evaluate the basic overhead of our monotonic casts for lambdas and records, we adapted the `sieve` benchmark from the benchmark suite by Takikawa et al. [2016] (also used in Section 5.8 and wrote a quicksort-based benchmark we call `intersort`, which implements a sorting library for integer list. The interface of the sorting library is specified using interfaces for iterators, mutable doubly recursive lists, and mutable list nodes for those lists.

As in the original benchmarking scheme by Takikawa et al. [2016], each program is divided up into modules – for each module, there exist different versions. To run a benchmark, we run all possible combinations of the different versions to get all possible kinds of mixing typed and untyped code, at least at a somewhat coarse granularity. Usually, each module just has a typed and an untyped version, but here we also are interested in the transition from structural to nominal. Technically, this would give us two degrees of freedom: typed vs. untyped and structural vs. nominal. However, because of the restrictions on method overriding discussed in Section 6.4.3, an untyped nominal module may be incompatible with a typed nominal module, as one might contain a specification of class or interface that extends a class or interface in the other. In Nom, the calculus and implementation implicitly made things typed and added casts in many cases, but MonNom does not do that, so we omit the untyped nominal modules from consideration. This leaves us with three versions for each module:

1. "Typed Nominal" (or just "Nominal"): fully typed and nominal, i.e. there exist no structural values whatsoever. For example, the

2. "Typed Structural": replacing nominal classes with structural values (i.e. lambdas or structures) where appropriate, but retaining type annotations where applicable; we explain for each benchmark below what that means for each benchmark

3. "Untyped Structural": like typed structural, but with all type annotations everywhere erased

For example, "Untyped Structural" version of the benchmark `Sieve` contains the following lambda:

```
() => Main.CountFrom(n+(dyn)1)
```

The corresponding "Typed Structural" version of `Sieve` adds type annotations everywhere and removes the artificial casts to **`dyn`** - including the lambda:

```
() => Main.CountFrom(n+1) : Stream
```

Finally, the "Nominal" version of `Sieve` instead contains the code

```
new CountFromFun(n)
```

where the class `CountFromFun` is implemented as follows:

```
class CountFromFun implements Fun<Stream>
{
  private readonly Int N;
  public constructor(Int n)
  {
    N=n;
  }
  public fun this() : Stream
  {
    return Main.CountFrom(N+1);
  }
}
```

Both benchmarks were run on an Intel Core i7-3770 at 3.4Ghz with 16GB of main memory, running Windows 7 with minimal background activity. The reported running times represent the average over 10 runs of the benchmark.

6.8.1 *Sieve*

As before, `Sieve` consists of two modules, yielding 9 possible configurations. It implements the Sieve of Eratosthenes and still causes catastrophic overhead for Typed Racket [Tobin-Hochstadt and Felleisen, 2008], though Grift has managed to reduce the overhead from catastrophic factors to just significant factors [Kuhlenschmidt, Almahallawi, and Siek, 2019]. The only difference between Nominal and Typed Structural here is that the nominal classes implementing the functional interfaces for the three continuations that create the sieving stream are replaced with lambdas (see example above). All other code is mostly not based on objects and instead in static methods; technically, the stream implementation could be re-worked to be based on structs, but we would then also need to introduce interfaces for it, and in general this is not the point of this benchmark.

Figure 6.10 shows the results of running this benchmark. On the right, there is basically no difference between the configurations where the main implementation is nominal and the stream library is either nominal or typed structural, which makes sense because the `Stream` module contains no lambdas, so those configurations are identical. These configurations are about 30% faster than the fully untyped structural configuration on the bottom. The slowest configuration, where the main implementation is nominal but the stream library is untyped structural, is about 7% slower than the fully untyped configuration, which is a reasonable amount of overhead and in line with our results from Nom. The relative slowdowns in this configuration compared to the others mostly comes from the fact that interacting with a lambda that has been cast is a little bit more expensive

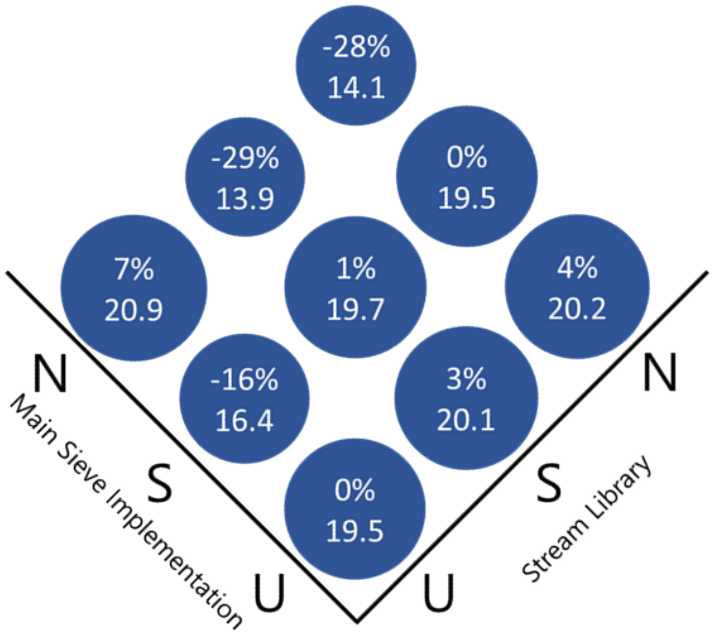


Figure 6.10: Results for the various configurations of `Sieve`. Each axis is a module, and their versions are labeled *U* (Untyped Structural), *S* (Typed Structural), or *N* (Nominal). The area of the bubbles is proportional to their running times, and contain the overhead compared to the fully untyped structural configuration at the bottom in percent and the absolute running time in seconds. A more classical bar graph with error bars can be found in Appendix C.1.

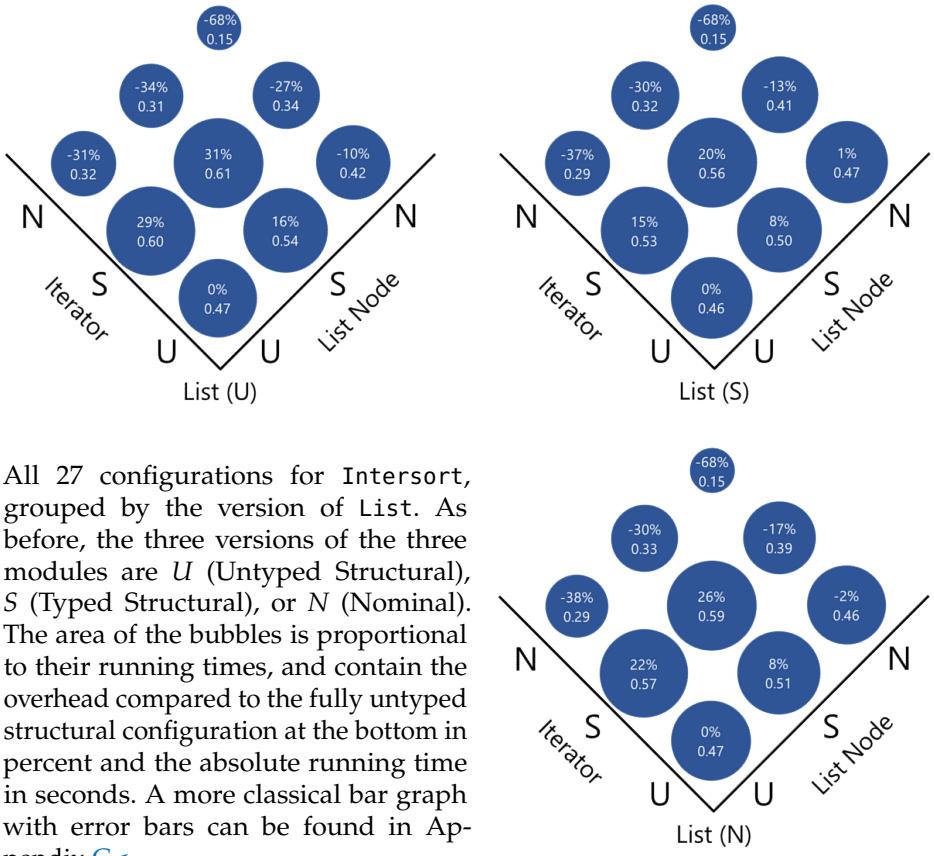
for untyped code, while due to the tight coupling of streams with the main implementation code not a lot of other typed-based optimization potential exists. In absolute numbers, the running times very closely match those of Nom (compare to Figure 5.10).

6.8.2 *Intersort*

`Intersort` consists of one always nominally typed sorting module and three modules (`Iterator`, `List`, and `ListNode`) in different versions, yield-

ing 27 possible configurations. We designed this benchmark to specifically evaluate the overheads of switching between object-oriented structural and nominal patterns. It implements the quicksort algorithm on a mutable doubly-linked list and per run sorts a list of 200,000 integers. The difference between Nominal and Typed Structural here is as follows: the nominal code was designed to not directly call the constructors of the implementing classes for iterators, lists, and list nodes – instead, there are static methods implemented in each module that essentially wrap those constructors. Then for the Typed Structural version, we delete the classes for iterators, lists, and list nodes, and instead in the static constructor methods construct the corresponding records. Since in Typed Structural the static constructor method is typed (as returning an instance of the appropriate interface), the records are cast immediately after being constructed. In Untyped Structural, where the type annotations are removed (at least from the module that contains the static constructor method), those casts may happen later.

Figure 6.11 shows the results of running the benchmark. Here, we see that structural code is indeed a lot slower than nominal code, even without the overhead from gradual typing. The fully untyped, structural configuration UUU takes about three times as long to run as the fully typed, nominal configuration NNN. The configurations in between vary a lot, in part because the figure now does not represent a linear gradient from untyped to typed anymore, as there are three different versions of each module. We see that the middle module `List` has the least influence on the running time, which makes sense, as it is only used to kick off the first level of quicksort; most of the work is done by the first module, `Iterator`, which in turn often accesses `ListNodes`, implemented in the last



All 27 configurations for `Intersort`, grouped by the version of `List`. As before, the three versions of the three modules are *U* (Untyped Structural), *S* (Typed Structural), or *N* (Nominal). The area of the bubbles is proportional to their running times, and contain the overhead compared to the fully untyped structural configuration at the bottom in percent and the absolute running time in seconds. A more classical bar graph with error bars can be found in Appendix C.1.

Figure 6.11: Results for the various configurations of `Intersort`.

module. The maximum overhead is about 31%, and overall, the worst-performing configurations are those where `Iterators` and `ListNodes` are typed structural. This is because the typed structural version still contains structs, but also many type annotations that force these structs to be cast, and casting structs has fewer optimization potential than lambdas. While those overheads are a lot more than for `Sieve`, they are still a lot lower than the multiples of running times all previous work on sound gradual typing suffered from. This suggests that structs can, in the use cases we envision, effectively masquerade as interfaces, and that the transition from

untyped structural to typed nominal code can be done without suffering catastrophic overheads in the meantime.

6.9 SUMMARY

In this chapter, we saw a formal development of how to reason about transitioning not only from untyped to typed code, but also from structural code to nominal code. However, our implementation of a language featuring this concept shows reasonably efficient casts for microbenchmarks aimed at stress-testing the core use cases. As such, this is a major stepping stone in lifting the restrictions of Nom to obtain sound, efficient, and convenient gradual typing.

7

DISCUSSION

The languages discussed in the previous two chapters are the currently best-performing sound gradually typed languages in terms of the overhead of gradual typing. This performance comes at the price of expressiveness; the languages are not yet as feature-rich as the other languages out there. As such, they provide a baseline for efficiency from which more expressive sound gradual type systems can be explored while trying to keep the overhead minimal. In this section, we argue that this baseline is valid and useful, sketching an outlook of how to get to more expressive type systems from it, and sketching the challenges that still need to be overcome.

7.1 DESIGNING FOR PERFORMANCE

Nom's and MonNom's efficiency comes from a combination of multiple factors that keep potentially expensive run-time operations cheap. Nominality makes casts infrequent and efficient. Transparency for nominal values prevents overhead due to wrapper allocation, and the non-transparent casts in MonNom still avoid as much overhead as possible. Immediate accountability makes blame tracking unnecessary – at least in Nom; MonNom loses this property, as lambdas and structs may have to delay some checks. However, it is common practice to deactivate blame tracking by default and only enable it when a program is being debugged [Vitousek,

Swords, and Siek, 2017]. Furthermore, the clear separation of dynamic vs. statically checked field accesses and method invocations allows us to implement and optimize both using the techniques appropriate to each. In the following, we illustrate the advantages of each of these properties by comparing Nom/MonNom to the other languages that we benchmarked.

TYPED RACKET In Typed Racket, most of the overhead of gradual typing is caused by expensive run-time checks. Compared to Nom/MonNom, these have two major causes: wrappers vs. transparency, and structural vs. nominal. In gradually-typed Racket, transferring a value across the typed/untyped boundary, in either direction, often requires the value to be wrapped in order to enforce soundness and provide blame. Thus each such transfer causes an allocation, and these wrappers themselves often have to produce more wrapped values, leading to more allocations. Allocation is known to be a fairly slow operation even in Racket where allocation is quite optimized. Furthermore, these wrappers introduce layers of indirection, especially since wrappers often end up being stacked onto each other in our benchmarks. Note that this stacking is indirect, so using *threesomes* à la Siek and Wadler [2010] would not help—in fact it would only add more overhead (indeed, Feltey et al. [2018] found no improvement for the `sieve` benchmark when trying to combine wrappers). Our systems are transparent, so we suffer none of these allocations or layers of indirection.

The second major reason is that Typed Racket uses a structural type system whereas we use a nominal type system. In Typed Racket, one can dynamically check that a value has a field of the appropriate name, and one can check that the value in that field is a function. However, one cannot

check what kind of function it is. Consequently, every time one uses that function to get an integer (the type that your typed code is expecting), one has to check that it actually returns an integer. This is even the case when that function was created by typed code but happened to transfer through untyped code. In Nom (and the nominal parts of MonNom), we can often accomplish all these checks with a single nominal check In particular, if the value was created by typed code, then that single nominal check accomplishes what Racket would need a potentially infinite number of structural checks for. For the structural parts of MonNom, we can still avoid the extra checks for values created by typed code. In problematic programs, such as `sieve` and `snake`, these two major differences can each introduce multiple factors of overhead in gradually-typed Racket (but not in Nom or MonNom), explaining the performance differences (the fact that MonNom has some structural features yet performs reasonably suggests that wrapper allocation may be the bigger factor).

C# We experimented with a few variations of dynamic programs in C# to investigate why increased dynamism causes large overheads. As an example, we experimented with casting everything to and from `Object` instead of **`dynamic`**. Conceptually, these two programs should have the same performance since the **`dynamic`** program is simply doing those casts implicitly at run time. However, we found that the `Object` version of the program was significantly faster. This leads us to believe that C#'s choice of implementation of **`dynamic`**, which recompiles the relevant code at run time using the run-time type of the relevant value, is the cause of its inefficiency. This choice seems to be forced upon C# in order to accommodate the many type-system features of C# that were not designed

with gradual typing in mind, an issue we will discuss this further in Section 7.3.

RETICULATED PYTHON As already stated in Section 5.8.3.3, the transient casting strategy in Reticulated Python inserts more casts as more type annotations are added, making fully typed code the slowest configuration of any program. In contrast, pessimistically typed code in (Mon)Nom is sound without any casts being inserted and can additionally benefit from type-directed compiler optimizations. Furthermore, the numbers we give for the Reticulated Python benchmarks are for the versions of the programs where blame tracking was turned off. Blame tracking significantly increases—sometimes doubles—the overhead of gradual typing in Reticulated Python. In contrast, as discussed in Section 5.7.2, Nom's immediate accountability makes blame tracking completely unnecessary – we have not yet measured the overhead of blame tracking in MonNom.

7.2 SCALING TO INDUSTRY

Our compilers and languages are experimental and thus lack many features that real-world compilers and languages would have, such as support for debugging, multithreading, separate compilation, etc. However, features that do not affect the type system should not affect the efficiency of gradual-typing-related operations. In fact, in contrast to some systems with monotonic run-time type information, Nom's approach has trivial multithreading support, as all operations during a cast are read-only. MonNom has been designed to support multithreading in an efficient way,

but we so far lack the benchmarks to say anything conclusive about how well this would work in practice. With respect to type-system features, the major differences between (Mon)Nom and pre-generics Java is that (Mon)Nom restricts overloading and does not support exceptions and (in Nom's case) arrays (Nom provides a natively implemented `ArrayList` class whose getter method is typed as returning `dynamic`). We do not believe that adding these features would cause significant gradual-typing overhead. If that turns out to be the case, then Nom and MonNom should be easy to extend to something that could be used in an industrial setting, although incorporating more powerful features such as variant generics will still require some more work, as discussed next and in Chapter 8.

7.3 INCREASING EXPRESSIVENESS

Nominal typing faces many challenges specific to nominality. This is true even without gradual typing, and still today discoveries about the foundations of nominal typing are being made. Here we discuss some of the challenges related to gradual typing, in particular ones that make the gradual guarantee difficult to achieve.

7.3.1 *Types Affect Execution*

Unlike structurally typed languages, types in statically typed nominal languages often affect execution. Examples of this are method overloading and extension methods. With method overloading, the type of the arguments is used to determine which overloading to call. When there is

ambiguity, languages like C# and Java report a type error forcing the programmer to resolve the ambiguity before compiling. This use of ambiguity as error is often necessary when types affect execution in order to keep execution predictable.

However, with gradual typing these ambiguities arise at run time, when the programmer is not generally available to disambiguate the execution, and so an error is thrown. And because the run-time types of values can provide more overloadings than might have been statically available, making a program more dynamic can even introduce such ambiguity errors, violating the gradual guarantee. Thus, gradually typed nominal languages cannot rely on the programmer to resolve the many ambiguities that statically typed nominal language often have. For method overloading, this means all overloadings provided by a given class or interface must have pairwise disjoint signatures in order to satisfy the gradual guarantee.

C# has another issue with gradual typing: extension methods. Extension methods are a way to retroactively add methods to an interface or class. In C#, when the type-checker fails to find a method declaration in a given receiver's class or interface, it checks for extension methods defined for that class or interface in the current static scope. But this faces two challenges when gradually typed. First, the current static scope is not available at run time. As a consequence, C# fails to identify extension methods at run time and simply throws a run-time type exception. This means that LINQ, the specialized syntax used to describe database queries that is built entirely on extension methods, is completely unusable by dynamically typed C#. Second, a method declaration that is not visible at compile time, so that the extension method is invoked, could be visible at run time, so that the

instance's method is invoked, thereby causing dynamic typing to change the semantics of the program, violating the gradual guarantee.

7.3.2 *Generics*

Java, C#, and Scala have all had generics for over a decade, and more recent nominal languages such as Ceylon and Kotlin continue the trend. Thus gradual typing for nominal types needs to address generics. Ina and Igarashi have considered gradual typing for generics [Ina and Igarashi, 2011]. They have an interesting approach to `Foo⟨dynamic⟩`, where `Foo⟨T⟩` is a generic class, which is to consider all uses of `T` in the body of `Foo` as potentially **`dynamic`**. Unfortunately, this means that even well-typed code may need to have frequent run-time checks inserted. Furthermore, they do not consider generic methods, type-argument inference, or any form of variance, all of which are essential to how generics are used in practice. As such, expressive generics and sound gradual typing have not yet been successfully married. The reason for that lies in a number of challenges.

The first challenge was decidability of subtyping in general, which we covered Part I. With respect to efficiency, it is important to note that our approach requires the ability to check subtyping at run time. This implies that every instance of `List⟨String⟩` stores the information necessary to determine at run time that the instance is not just a `List`, but a `List` of `String`. This is known as reified generics and is the counterpoint to type erasure. This might cause some concern, as reified generics imply that type information has to be constructed and passed around at run time throughout generic methods. Schinz [2005] did an analysis of what the

impact of this would be for Scala on the JVM, and he found that it would on average make programs run 50% slower, allocate 140% more memory, and compile to 30% more byte code. However, Microsoft added reification to the CLR because of its potential to improve performance with primitive types and specialization, and Ceylon's generics are reified because the team found they could implement it with little overhead, even on the JVM. It is thus unclear what the overhead of reified generics might actually entail for gradual typing.

The greatest challenge, though, is likely to be type-argument inference, a feature that is critical to making use of generic methods convenient. To understand why it is likely to be such a significant challenge, consider the following statically typed C# function `SnocS` (without extension methods):

```
List⟨T⟩ SnocS⟨T⟩(IEnumerable⟨T⟩ startS, T endS) {
    var elemsS = Enumerable.ToList(startS);
    elemsS.Add(endS);
    return elemsS;
}
```

and its corresponding dynamically typed C# function `SnocD`

```
dynamic SnocD(dynamic startD, dynamic endD) {
    var elemsD = Enumerable.ToList(startD);
    elemsD.Add(endD);
    return elemsD;
}
```

In order to fulfill the gradual guarantee, if a call to `SnocS` succeeds, the same call to `SnocD` should succeed. However, the calling the typed version `SnocS(new List⟨String⟩(), 5)` succeeds in C#, whereas the untyped version `SnocD(new List⟨String⟩(), 5)` throws a run-time type exception.

In particular, the invocation `elemsD.Add(endD)` fails because at run time `elemsD` is a `List⟨String⟩` but `endD` is an `Int`. In the corresponding line of `SnocS`, the run-time type of `elemsS` is `List⟨Object⟩`. The cause of the difference in behavior is that in `SnocS`, the type argument for `ToList` is inferred to be `T`, which at run time is `Object`, due to the static type of `startS` *and* `endS`, whereas in `SnocD` it is inferred to be `String` due to the dynamic type of `startD`. Thus, in addition to developing a decision procedure for type-argument inference, a gradual type system for generics must also overcome this challenge regarding the gradual guarantee. We discuss this further in Chapter 8.

It is due to these many complications with nominal typing that C# is forced to implement gradual typing using run-time compilation. This unfortunate fact is likely the cause of its poor performance in Section 5.8. Thus, with nominal typing, it seems to be important to design the language with gradual typing in mind in order to not only achieve the gradual guarantee, but also to achieve efficient implementation of dynamic typing.

Part III

GENERICS

8

TOWARDS INFERABLE AND GRADUALIZABLE GENERICS

8.1 INTRODUCTION

We already discussed the need for decidability in gradual typing and its usefulness in general in Chapter 1. However, as we have mentioned many times in the preceding chapters, decidability is not all that is needed for gradual typing to be well-behaved. In languages with generics, type-argument inference for generic methods tends to be the primary culprit for unpredictable changes in program behavior because of changes in type information. Due to subtyping, there is no known sound and complete algorithm for type-argument inference, so compilers are forced to employ ad-hoc heuristics. Plus, again due to subtyping, there can multiple valid type-arguments, which has semantic significance in *reified* languages, meaning type arguments can be examined at run time, which we already mentioned is necessary to do run-time type checks for gradual typing. Thus whether a program type-checks often varies by compiler, and even how that program executes can vary by compiler. This is a problem for the program-writer because they can unwittingly get locked into a particular

compiler[1], and it is a problem for the compiler-writer because they can unwittingly get locked into a particular type-checking algorithm[2].

These problems with type-argument inference become all the more pronounced by gradual typing. Gradual typing must perform many type-checking operations at run time in order to provide sound behavior. Consequently, the unreliability and inconsistency that type-argument inference has traditionally only caused at compile time now become run-time problems. Thus, before one can even consider developing a *practical* gradually typed language for generics, one first needs to provide the infrastructure for supporting run-time casts and for reliable and consistent dynamically typed invocations of generic methods. That is, one must first make generics *gradualizable*. While here we have made it clear that decidability affects gradualizability, later in the chapter we will demonstrate that in fact gradualizability can also affect decidability.

This chapter strives towards making generics decidable (and even locally inferable) and gradualizable while still being practical. In particular, we selected the features of modern major languages that seem to be the most important to the usability of generics: inheritance, mixed-site variance, type-argument inference, local lambdas, reification, and a solution to the binary-method problem. We then developed two calculi that provide those features while simultaneously being decidable and (mostly) gradualizable, taking a major step towards principled practical generics.

1 This problem grew significant enough that for Java 8 there was a concerted effort across the compiler teams along with the language team to develop essentially an official type-checking algorithm, particularly for type-argument inference.

2 After improving type-argument inference for Java 8, Oracle discovered that the more-precise types being inferred were changing overload resolutions, altering the semantics of preexisting programs [Oracle Corporation, 2014, Area: Tools/`javac`].

8.2 OVERVIEW

The primary challenge of this chapter is the many interacting features that need to be considered simultaneously. Rather than first present all these features and their full myriad of interactions, we instead illustrate a few issues they cause in C#, the best-known language that supports all of them.

8.2.1 *The Binary-Method Problem, Declaration-Site Variance, and Decidability*

Designs for generics inevitably run into the binary-method problem, with type-safe equality being the most pressing case. For many designs, one wants to be able to check if the contents of two variables are equal in some deep sense rather than just shallow reference equality. However, in order for, say, a `String` to check if it describes the same character sequence as some other value, it first needs to know that that other value is also a `String` and so has the fields it needs to inspect. It can do so for any `Object` by using a dynamic cast, but that means the type-checker will fail to warn the programmer that they wrote `str.Equals(person)` rather than `str.Equals(person.Name)`. Thus this form of equality is not considered to be (statically) type-safe.

The obvious alternative is to have `String`'s `Equals` method accept only `String`s. But this introduces a new problem: many generic data structures need to use equality but can no longer assume that all values are equatable with all other values. That is, in order for these libraries to be *parametrically* polymorphic over a type variable `E`, they need some way to *constrain* `E` so

that they can ensure that values of type α are at least equatable with each other.

F-bounded polymorphism was designed for precisely this purpose [Canning et al., 1989]. One designs a generic interface `Eq⟨T⟩` that guarantees a method `Equals(T)`, and then the library simply requires that `E` be a subtype of `Eq⟨E⟩`.

Unfortunately, this solution quickly encounters its own limitations. For example, one would like to be able to equate sequences (i.e. read-only lists) with other sequences, but this is only possible when the values these sequences contain are themselves equatable with each other. That is, equatability of sequences is *conditional* upon equatability of their contents.

Ideally one could combine *declaration-site* variance with inheritance to address this problem. Something that is equatable with `Number` is also equatable with `Integer` since every `Integer` is a `Number` and therefore can be passed to the appropriate `Equals` method. This makes `Eq` a *contra*variant interface, which C# supports by adding **in** to the *declaration* of the `Eq` interface, as in `Eq⟨`**in** `T⟩`. On the other hand, a sequence of integers is also a sequence of numbers, making sequence a *co*variant interface. This is expressed in C# via the declaration `Seq⟨`**out** `E⟩`. If one could furthermore declare that `Seq⟨E⟩` extends `Eq⟨Seq⟨Eq⟨E⟩⟩⟩`, then the combination of co- and contravariance would actually make it so that `Seq⟨E⟩` is equatable with itself whenever `E` is.

This would successfully encode conditional equatability for sequences if it were not for the fact that it introduces yet another problem. Suppose the class `Tree` implements `Seq⟨Tree⟩` to provide its sequence of children. Consider whether `Tree` is equatable with itself. Trees are equatable if sequences of trees are equatable, and sequences of trees are equatable

if trees are equatable. That is, trees are equatable if and only if they are equatable through conceptually cyclic reasoning. While in this case the cycle is easy to identify, in the more general case the cycle can be irregular and undetectable, making subtyping undecidable [Grigore, 2017; Kennedy and Pierce, 2007].

For this reason, C# disallows *expansive* inheritance clauses, which ensures that all cycles can be detected and makes subtyping decidable [Kennedy and Pierce, 2007]. But having `Seq⟨E⟩` extend `Eq⟨Seq⟨Eq⟨E⟩⟩⟩` is an expansive inheritance clause, and so this restriction also blocks the above solution to conditional equatability. As such, even new major languages with generics still fall back on the aforementioned casting mechanism for equality rather than a type-safe mechanism.

8.2.2 *Type-Argument Inference*

Type-argument inference is critical to the usability of generics. Unfortunately, it is also complicated. Consequently, following the research convention of making calculi as simple as possible, most calculi for generics require all type arguments to be explicitly specified. When verifying soundness, such as in Featherweight Generic Java [Igarashi, Pierce, and Wadler, 2001], this convention is perfectly valid. But when designing for gradual typing, such as in the extension alluded to by Ina and Igarashi [2011], this convention fails to address the greatest challenge in gradualizing generics. Here we provide a glimpse of that challenge, using the following methods and assuming variables `b1` and `b2` whose respective class types `B1` and `B2` both extend class `A`.

```
Seq⟨E⟩ Singleton⟨E⟩(E elem) {
    var list = new List⟨E⟩(); list.add(elem); return list;
}
List⟨E⟩ Snoc⟨E⟩(Seq⟨E⟩ head, E tail) {
    var l = new List⟨E⟩(head); l.add(tail); return l;
}
T Random⟨T⟩(T first, T second) {
    if (CoinFlip()) return first; else return second;
}
```

In C#, the expression `Snoc(Singleton(b1), b1)` has type `List⟨B1⟩`. In particular, the type arguments for the generic methods `Singleton` and `Snoc` are both inferred to be `B1`. While that inference works well for this expression, it is not necessarily ideal for the context of the expression.

Suppose we want to add `b2` to the resulting list. We cannot do so given the inferred type of the list because it only accepts `B1`s. If we had instead declared `b1` to have the *super*type `A`, voluntarily losing precision of type information, then the inferred type of the list would instead be `List⟨A⟩`, which we could add `b2` to. This observation means that any language that infers the type arguments in `Snoc(Singleton(b1), b1)` necessarily fails to ensure *subsumption*, a property of subtyping.

For many languages, subtyping has meaning at run time, guaranteeing that *values* of the subtype can be used wherever values of the supertype are acceptable. Subsumption is the property that subtyping also has meaning at compile time, guaranteeing that *expressions* of the subtype can be used wherever expressions of the supertype are acceptable. That is, subsumption means that sub*typing* has a formal connection to program *typing* rather than simply being an approximation of sub*set* that is ad-hocly incorpo-

rated into type-checking. Unfortunately, none of the major languages with generics ensure subsumption, with the ad-hoc nature of type-argument inference being a major obstacle to achieving subsumption.

Failing to ensure subsumption is particularly problematic for gradual typing. Suppose we first assign `b1` to a variable `a` of type `A`, and then we use `Snoc(Singleton(a), a).Add(b2)` instead. This will type check in C# and execute as expected. But if we change the type of `a` to be **`dynamic`** instead, utilizing C#'s support for gradual typing, then it will type-check the code using the *dynamic* type of `a`, which will be `B1`, and this will cause the subsequent invocation of `Add` to fail. This fickleness in the semantics means that C# fails to satisfy the (dynamic) gradual guarantee, which in short asserts that adding or removing correct type annotations should not change the semantics of programs. Thus, in order to support well-behaved gradual typing, type-argument inference needs to be guaranteed to be more successful as more-precise type information is made available.

8.2.3 *Principal Types*

Notice that the above problem with subsumption was entirely due to `Snoc`. This is because `Singleton` has the property that, no matter what argument is provided, there is always a type argument that results in a *principal* type for the invocation. A language has principal types if every typeable expression can be given a most-precise type (also known as minimal types [Balsters and Fokkinga, 1991; Bruce, Crabtree, and Kanapathy, 1994; Ghelli and Pierce, 1998], and not to be confused with principal type *schemes* [Damas and Milner, 1982; Hindley, 1969] such as in ML). If the

language ensures subsumption, then this means every typeable expression can be given a type that is a subtype of all other ascribable types.

Principal types are great for decidability because they ensure that there is a single type that best describes the expression, meaning the compiler does not have to backtrack through multiple options (assuming there are only finitely many). They often even make various local forms of inference practical by recursively computing the principal types of expressions. Invocations of `Snoc` cannot always be given a principal type due to type-argument inference, but invocations of `Singleton` can, and as such we say it is (or its type arguments are) principally inferable.

Note that this is a property of how `Singleton` is *declared* rather than of how it is *used*. Type-argument inference has so far focused on uses of generic methods. By shifting the focus to declarations of generic methods we will be able to ensure principal typing. Furthermore, we will even be able to develop a new semantics for reified generic methods, one that will ensure gradual typing. But we first need to discuss the many obstacles to making this change in perspective practical.

8.2.4 *Semantic Coherence*

`Singleton` still poses problem for gradual typing despite being principally inferable. Suppose we assign `b1` to a variable `a` of type `A`, then we assign the expression `Singleton(a)` to a variable `list` of type **`dynamic`**, and then we execute `list.Add(b2)`. In C#, this will execute successfully because `Singleton`'s type argument will *statically* be inferred to be `A`, causing its invocation to allocate a `List`⟨`A`⟩, which will then accept `b2`. However, if

we change the type of `a` to be **`dynamic`**, then the type argument will be *dynamically* inferred to be `B1`, causing `Singleton` to allocate a `List⟨B1⟩`, which will reject `b2`, causing a run-time type error.

The issue is that the semantics of *statically* typed C# fails to be *coherent*. C#, like many statically typed languages, has a type-directed semantics, meaning the proof that the program type-checks is used to determine its run-time behavior. But, with a reasonably declarative specification of the type system, there are generally *many* ways to prove that a program is well-typed. Coherence, then, is the property that all these declarative type-checking proofs result in the same run-time behavior. The issue with `Singleton` arises from the fact that the proof itself determines the type argument, and that type argument has semantic significance due to reification and the potential for casting (whether in the surface language or in the cast calculus for gradual typing). Because of subtyping, there are multiple valid type arguments for this invocation, causing the semantics to be fickle in the face of gradual typing, as illustrated. Note that gradual typing is not strictly necessary to expose this fickleness; one could simply attempt to cast `seq` to `List⟨A⟩` in statically typed C# to observe the same behavior. As with all the problems illustrated here, it is not limited to C#. Indeed, it seems to be fundamental to the standard semantics for reified generics, hence we need to develop a new semantics.

8.2.5 *Joins and Meets*

But before diving into semantics, we need to further understand the challenges of type-argument inference. Consider now the expression

`Random(x, y)`, where `x` has type `X` which extends `Seq⟨X⟩`, and `y` has type `Y` which extends `Seq⟨Y⟩`. C#'s inference algorithm intentionally fails in this case [ECMA TC39-TG2, 2017, §12.6.3]. The reason is that it requires computing a common supertype of `X` and `Y`, and doing so in a principled manner is impossible for C#.

To see why, consider what common supertypes `X` and `Y` have. We can see they are both sequences, and due to covariance of `Seq` they are in fact both subtypes of `Seq⟨Object⟩`. But better yet they are both subtypes of `Seq⟨Seq⟨Object⟩⟩`, and even better they are both subtypes of `Seq⟨Seq⟨Seq⟨Object⟩⟩⟩`. This increase in precision can go on forever, so the ideal answer, i.e. the least common supertype or *join*, would be the infinite type `Seq⟨Seq⟨Seq⟨...⟩⟩⟩`. Unfortunately, this type is not expressible, and consequently the join of `X` and `Y` does not exist.

This means that, not only does `Random(x, y)` not have a principal type, it does not even have a finite number of types that cover all of its possible typings. The lesson here is that, first, the subtyping system needs to ensure the existence of joins (and consequently *meets* due to contravariance) in order to provide principal types for many practical examples, and, second, C#'s rejection of expansive inheritance fails to ensure joins even though it ensures decidable subtyping.

8.2.6 *Ambiguity*

Moving beyond type-argument inference, we discuss one last issue that is common in statically typed languages. Many statically typed languages have situations in which ambiguities arise. For example, C# permits *mul-*

tiple-instantiation inheritance, meaning a class `Seqs` can simultaneously implement both `Seq⟨Int⟩` and `Seq⟨String⟩`. If `Seq⟨E⟩` has an `E First()` method, then `Seqs` can implement that method differently for its two instantiations of `E`. If one assigns a `Seqs` instance to a `Seq⟨Int⟩` variable, then it is clear how all method invocations on that variable should operate. However, if that variable were given the more precise type `Seqs`, then it is unclear how invocations of `First` should proceed. C# recognizes this ambiguity and fails to type-check the invocation, providing an error informing the programmer that they need to clarify, typically by upcasting the variable to the intended instantiation.

Many ambiguities in statically typed languages are of this form. The type information is used to determine the semantics, but there is too much type information in order to sufficiently guide this determination. Thus the programmer must explicitly remove type information from the situation. Any language with this behavior necessarily fails to exhibit subsumption.

This observation is particularly important regarding the binary-method problem. Recent proposals address binary methods by using either implicits [Odersky, Altherr, et al., 2014, Chapter 7] or constraints [Zhang, Loring, et al., 2015] to emulate type-class evidence [Wadler and Blott, 1989]. Manually specifying this evidence is tedious, and as such these proposals construct it automatically based on type information. However, because this evidence is semantically meaningful, they defer to the programmer when ambiguities exist, consequently failing to exhibit subsumption, which we will need for decidability and gradualizability.

$$
\begin{array}{lll}
\text{Hierarchy} & \Psi & ::= c; \ldots \\
\text{Expression} & e & ::= \text{(see Figure 8.9)} \\
\text{Confluence} & c & ::= c^I \mid c^S \\
\text{Interface} & c^I & ::= \text{(see Figure 8.2)} \\
\text{Shape} & c^S & ::= \text{(see Figure 8.5)}
\end{array}
$$

$$\boxed{\text{Program Validity } \vdash \Psi; e}$$

$$\frac{\vdash \Psi \qquad \Psi \mid \varnothing \mid \varnothing \mid \varnothing \vdash e : \top}{\vdash \Psi; e}$$

$$\boxed{\text{Hierarchy Validity } \vdash \Psi}$$

$$\frac{\vdash^I \Psi \qquad \Psi \vdash^S \Psi \qquad \Psi \vdash^\sigma \Psi \qquad \Psi \vdash^s \Psi}{\vdash \Psi}$$

Figure 8.1: Programs and Hierarchies (with rules for missing judgements to be found in Figures 8.2, 8.5, 8.6, 8.8, and 8.9)

8.3 INTERFACES AND SUBTYPING

We now begin presenting the design of our calculi that address the many problems just discussed. In addition to providing the features of practical generics, these calculi will further ensure subsumption, principal types, and semantic coherence. By doing so, even though the calculi will not themselves provide gradual typing, they will enable well-behaved gradual typing to be built on top of them.

The calculi are type-checked in multiple passes. Each pass establishes invariants that the subsequent passes rely upon. These invariants, more than anything else, are the key to achieving our goals. As such we will present the calculi in the order of these passes, highlighting the invariants they ensure along with how and why. Note that this means expressions will be presented last.

A program in our calculi is a hierarchy Ψ and an expression e, which are introduced in Figure 8.1. A hierarchy is simply an ordered list of, for lack of a better term, confluences c. A confluence is either an interface c^I

or a *shape* c^S; in terminology of Chapter 2, interfaces are just materials, and shapes are completely separate. For simplicity our calculi model classes as methods that generate objects, which are in turn simply closures implementing an interface. Shapes, on the other hand, will be how we address the binary-method problem. Rather than being part of types themselves, shapes will be constraints that types can *satisfy* – types will be comprised solely from (material) interfaces. Note that throughout this chapter that superscripts are used to distinguish related grammars—such as c^I for interface confluences versus c^S for shape confluences—and are *not* used to parameterize grammars, meaning the use of I in c^I places *no* restriction on which interface is defined by c^I.

Figure 8.1 shows that programs are type-checked in five passes, the first four of which validate the hierarchy. The first pass validates interfaces and ensures that subtyping is well-behaved. The second and third passes validate shapes and ensure that shape satisfaction is well-behaved. The fourth pass ensures that method signatures are well-behaved. And, with the hierarchy validated, the final pass ensures that the actual implementation is well-behaved.

Appendix D.3 contains the full grammar an index of every part of the static formalization. Most of it is specfied directly in the rest of the chapter as the components come up; a few very straightforward parts of the dynamic part of formalization are contained in Appendix D.4. In the rest of this section we discuss the first pass: interfaces and subtyping.

Interface Name	I	
Interface Declaration	c^I	$::= \texttt{interface}\ I\langle\Theta\rangle\ \texttt{extends}\ \vec{\tau}^I\ \texttt{satisfies}\ \vec{\sigma}^I\ \{s;...\}$
Kind Context	Θ	$::= \tau <: \nu\alpha <: \tau, ...$
Inherited Interfaces	$\vec{\tau}^I$	$::= \varnothing \mid \tau^I \mid \tau^I, ...$
Inherited Interface	τ^I	$::= I\langle\alpha, ...\rangle \mid I\langle\vec{\tau}\rangle$
Conditionally Satisfied Shapes	$\vec{\sigma}^I$	$::=$ (see Figure 8.6)
Method Signature	s	$::=$ (see Figure 8.8)
Signed Variance	$\nu^{\pm}$	$::= + \mid -$
Variance	ν	$::= \nu^{\pm} \mid !$
Ignorable Variance	$\nu^?$	$::= \nu \mid ?$
Type Variable	α	
Type	τ	$::=$ (see Figure 8.3)
Types	$\vec{\tau}$	$::= \tau, ...$

Interfaces Validity $\vdash^I \Psi$

$$\frac{}{\vdash^I \varnothing} \qquad \frac{\vdash^I \Psi}{\vdash^I \Psi; c^S} \qquad \frac{\vdash^I \Psi \qquad \Psi \vdash^I \langle\Theta\rangle \qquad \Psi \mid \Psi \mid \Theta \vdash^I \vec{\tau}^I}{\vdash^I \Psi; \texttt{interface}\ I\langle\Theta\rangle\ \texttt{extends}\ \vec{\tau}^I\ \texttt{satisfies}\ \bullet\ \{\bullet\}}$$

Interface Parameter Validity $\Psi \vdash^I \langle\Theta\rangle$

$$\frac{}{\Psi \vdash^I \langle\rangle} \qquad \frac{\Psi \vdash^I \langle\Theta\rangle \qquad \Psi \mid \Theta \vdash_- \tau_\ell \qquad \Psi \mid \Theta \vdash_+ \tau_u \qquad \Psi \mid \Theta \vdash \tau_\ell <: \tau_u}{\Psi \vdash^I \langle\Theta, \tau_\ell <: \nu\alpha <: \tau_u\rangle}$$

Interface Inheritance Validity $\Psi \mid \Psi \mid \Theta \vdash^I \vec{\tau}^I \qquad \Psi \mid \Psi \mid \Theta \vdash^I \tau^I$

$$\frac{}{\Psi \mid \Psi' \mid \Theta \vdash^I \varnothing} \qquad \frac{\Psi' = \Psi_I; \texttt{interface}\ I\langle\Theta_I\rangle\ \texttt{extends}\ \vec{\tau}_I^I\ \texttt{satisfies}\ \bullet\ \{\bullet\}; \Psi'' \qquad \Psi \mid \Psi_I \mid \Theta \vdash^I \vec{\tau}^I \qquad \Psi \mid \Theta \vdash_+ \langle\vec{\tau}\rangle : \langle\Theta_I\rangle \qquad \forall \tau_I^I \in \vec{\tau}_I^I . \exists \tau^I \in \vec{\tau}^I . \Psi \mid \Theta \vdash \tau^I <: \tau_I^I\lfloor\vec{\tau}/\Theta_I\rceil}{\Psi \mid \Psi' \mid \Theta \vdash^I \vec{\tau}^I, I\langle\vec{\tau}\rangle}$$

$$\frac{\Psi \mid \Theta \vdash_+ \tau^I}{\Psi \mid \Psi' \mid \Theta \vdash^I \tau^I}$$

Figure 8.2: Interfaces (with rules for missing judgements to be found in Figure 8.3)

8.3.1 *Interfaces*

An interface declaration c^I, as shown in Figure 8.2, specifies a kind context Θ of constrained type parameters α with variances ν, inherited interfaces $\vec{\tau}^I$, conditionally satisfied shapes $\vec{\sigma}^I$, and signatures s of methods guaranteed by the interface. In the first pass, we only aim to ensure that subtyping is well behaved, and as such here we focus on the type parameters and on interface inheritance. (Notice that shapes are completely ignored in this pass.)

The first step for validating an interface declaration (after checking the hierarchy up to this point) is to validate the constraints on the type parameters for the interface via the judgement $\Psi \vdash^I \langle\Theta\rangle$. The kind context Θ declares the type parameters, each with a lower bound τ_ℓ, an upper bound τ_u, and a variance v that can be either covariant ($+$), contravariant ($-$), or invariant (!). The rules of $\Psi \vdash^I \langle\Theta\rangle$ are designed to disallow recursive constraints on type variables; every type variable can only be constrained by types referencing preceding type variables. This is in line with material-shape separation and takes advantage of the fact that applications of F-bounded polymorphism will instead be addressed using shapes, discussed later. The rules also ensure that the lower bound is a subtype of the upper bound, which is useful for ensuring decidability (with transitivity) of subtyping within this kind context. In order to ensure that this subtyping check is itself decidable, all bounds are restricted to using interfaces preceding the one currently being validated, which have already been validated themselves.

The second step is to validate the inherited interfaces, the grammar and rules for which differ across the two calculi. The first calculus permits only single inheritance and only allows the type arguments to the inherited interface to be type variables. This is the calculus with the simplest type grammar that we could construct, and due to the latter restriction on inheritance we refer to the first calculus as Gen_{α}. The second calculus permits multiple inheritance and extends the type grammar with arbitrary intersection and union types as its solution for addressing the complexities that multiple inheritance introduces. As such, we refer to the second calculus as $\text{Gen}_{\cap}$. Nonetheless, unlike C#, $\text{Gen}_{\cap}$ does not allow arbitrary multiple-instantiation inheritance. Instead, its complex rules for interface inheritance $\Psi \mid \Psi \mid \Theta \vdash^{I} \vec{\tau}^{I}$ ensure *principal*-instantiation inheritance (see also Section 3.6.3), which will be discussed in more detail shortly.[3]

8.3.2 *Enforcing Constraints*

The differences between the two calculi are most significant in this first pass. This is because they take different approaches to ensuring that method invocations have a principal return type.

In order to understand the issue, consider the following hierarchy (where + indicates covariance):

```
interface Foo⟨+A⟩ { get() : A }

interface Bar⟨+A, +B <: A⟩ extends Foo⟨A⟩ { get() : B }
```

This hierarchy is conceptually valid because, although the return type of `Bar`'s `get` method differs from `Foo`'s, the subtyping constraint on `B`

3 The rules also cause all interfaces that would be indirectly inherited to be directly inherited, but that is just a formalism convenience and is not essential to calculus.

ensures that it is a more precise return type. Thus any implementation of an instantiation of `Bar`, which is by definition required to satisfy the type constraints, is necessarily also an implementation of the corresponding instantiation of `Foo`.

The issue, then, is the return type of `get()` when invoked on an expression of an arbitrary $\texttt{Foo}\langle\tau_A, \tau_B\rangle$. If we use `Foo`'s signature, then it will be τ_B, but if we use the supertype `Bar`'s signature, then it will be τ_A. One might think τ_B must be a subtype of τ_A due to the constraint on `Bar`'s `B`. However, it has been shown that enforcing these constraints in type validity is actually unnecessary for *soundness* [Amin and Tate, 2016; Tate, 2013]; it is only necessary when validating the declared type arguments of an *implementation* of an interface. So while our observation here suggests that these constraints might be useful for *decidability*, constraint enforcement is itself a design *choice*. Given that decidability is a primary goal of this work, both our calculi choose to enforce constraints.

8.3.3 *Intersection Types*

Enforcing constraints does not necessarily entirely solve the problem with return types though. A similar issue can arise in the presence of arbitrary intersection types. If interfaces `Biz` and `Baz` both happen to have a method with the same name but with incomparable return types, then there are multiple incomparable return types when invoking that method on an expression of type $\texttt{Biz} \cap \texttt{Baz}$. For this reason, Gen_α opts to forgo all forms of intersection types.

As we will see shortly, Gen_{α} must impose some significant restrictions to ensure decidability without intersection types, and so $\text{Gen}_{\cap}$ instead solves the problem with return types another way. Recall that we demonstrated the importance of semantic coherence to gradual typing, specifically to the (dynamic) gradual guarantee. Semantic coherence also ensures another property that was first observed in designing Forsythe [Reynolds, 1988, 1997]: if a particular method invocation can soundly be given two different types, then it can also soundly be given the intersection of those types (which in turn is a subtype of both other typings). This is because semantic coherence means the invocation would result in the same value regardless of how it is typed, and so the resulting value must simultaneously have both ascribable types, and consequently belongs to their intersection. Thus, while arbitrary intersection types cause problems with ensuring a principal method signature, when combined with semantic coherence they ironically can solve the very problem they create.

8.3.4 *Joins and Meets*

Joins are extremely useful for decidable type-checking. Their most common application is in combining the types of the two cases of a conditional expression. With generics, they are also useful in inferring a type argument for which multiple lower-bound constraints have been identified. And in the presence of contravariance, joins necessitate meets, which are also needed for inferring type arguments in certain less-common situations.

The existence of joins and meets depend very much on the grammar and validity of types and the definition of subtyping, presented in Figure 8.3.

Type	$\tau \ ::= \bot \mid \top \mid I\langle\vec{\tau}^a\rangle \mid \alpha \mid \tau \cup \tau \mid \tau \cap \tau$
Type Arguments	$\vec{\tau}^a \ ::= \tau^a, \ldots$
Type Argument	$\tau^a \ ::= \tau \mid \textbf{in}\ \tau\ \textbf{out}\ \tau$
Invariant Kind Context	$\Theta^! \ ::= \tau <: !\alpha <: \tau, \ldots$

Variance $\nu * \nu^?$

$\nu * \nu^?$	!	+	−	?
!	!	!	!	?
+	!	+	−	?
−	!	−	+	?

Subvariance $\vdash \nu \leq \nu^?$

$$\frac{}{\vdash ! \leq \nu^?} \qquad \frac{}{\vdash \nu \leq \nu} \qquad \frac{}{\vdash \nu \leq ?}$$

Type Validity and Variance $\Psi \mid \Theta \vdash_{\nu^?} \tau \qquad \Psi \mid \Theta \vdash_{\nu^?} \tau^I$

$$\frac{}{\Psi \mid \Theta \vdash_{\nu^?} \bot} \qquad \frac{}{\Psi \mid \Theta \vdash_{\nu^?} \top} \qquad \frac{\Psi \mid \Theta \vdash_{\nu^?} \tau_1 \quad \Psi \mid \Theta \vdash_{\nu^?} \tau_2}{\Psi \mid \Theta \vdash_{\nu^?} \tau_1 \cup \tau_2} \qquad \frac{\Psi \mid \Theta \vdash_{\nu^?} \tau_1 \quad \Psi \mid \Theta \vdash_{\nu^?} \tau_2}{\Psi \mid \Theta \vdash_{\nu^?} \tau_1 \cap \tau_2}$$

$$\frac{\bullet <: \nu\alpha <: \bullet \in \Theta \quad \vdash \nu \leq \nu^?}{\Psi \mid \Theta \vdash_{\nu^?} \alpha} \qquad \frac{\textbf{interface}\ I\langle\Theta_I\rangle\ \textbf{extends}\ \bullet\ \textbf{satisfies}\ \bullet \in \Psi \quad \Psi \mid \Theta \vdash \langle\vec{\tau}^a\rangle \rightsquigarrow \langle\vec{\tau}\rangle \mid \Theta^! \quad \Psi \mid \Theta, \Theta^! \vdash_{\nu^?} \langle\vec{\tau}\rangle : \langle\Theta_I\rangle}{\Psi \mid \Theta \vdash_{\nu^?} I\langle\vec{\tau}^a\rangle}$$

Type-Argument Validity $\Psi \mid \Theta \vdash_{\nu^?} \langle\vec{\tau}\rangle : \langle\Theta\rangle$

$$\frac{}{\Psi \mid \Theta \vdash_{\nu^?} \langle\rangle : \langle\rangle} \qquad \frac{\Psi \mid \Theta \vdash_{\nu^?} \langle\vec{\tau}\rangle : \langle\Theta'\rangle \quad \Psi \mid \Theta \vdash_{\nu*\nu^?} \tau \quad \Psi \mid \Theta \vdash \tau_\ell\lfloor^{\vec{\tau}}/_{\Theta'}\rceil <: \tau \quad \Psi \mid \Theta \vdash \tau <: \tau_u\lfloor^{\vec{\tau}}/_{\Theta'}\rceil}{\Psi \mid \Theta \vdash_{\nu^?} \langle\vec{\tau}, \tau\rangle : \langle\Theta', \tau_\ell <: \nu\alpha <: \tau_u\rangle}$$

Type-Argument Capture $\Psi \mid \Theta \vdash \langle\vec{\tau}^a\rangle \rightsquigarrow \langle\vec{\tau}\rangle \mid \Theta^!$

$$\frac{}{\Psi \mid \Theta \vdash \langle\rangle \rightsquigarrow \langle\rangle \mid \varnothing} \qquad \frac{\Psi \mid \Theta \vdash \langle\vec{\tau}^a\rangle \rightsquigarrow \langle\vec{\tau}\rangle \mid \Theta^!}{\Psi \mid \Theta \vdash \langle\vec{\tau}^a, \tau\rangle \rightsquigarrow \langle\vec{\tau}, \tau\rangle \mid \Theta^!}$$

$$\frac{\Psi \mid \Theta \vdash \langle\vec{\tau}^a\rangle \rightsquigarrow \langle\vec{\tau}\rangle \mid \Theta^! \quad \Psi \mid \Theta \vdash \tau_i <: \tau_o}{\Psi \mid \Theta \vdash \langle\vec{\tau}^a, \textbf{in}\ \tau_i\ \textbf{out}\ \tau_o\rangle \rightsquigarrow \langle\vec{\tau}, \alpha\rangle \mid \Theta^!, \tau_i <: !\alpha <: \tau_o}$$

Figure 8.3: Types and Subtyping

Named Type-Argument Capture $\vdash \langle \vec{\alpha}^! \rangle := \langle \vec{\tau}^a \rangle \rightsquigarrow \langle \vec{\tau} \rangle \mid \Theta^!$

$$\frac{}{\vdash \langle\rangle := \langle\rangle \rightsquigarrow \langle\rangle \mid \varnothing} \qquad \frac{\vdash \langle \vec{\alpha}^! \rangle := \langle \vec{\tau}^a \rangle \rightsquigarrow \langle \vec{\tau} \rangle \mid \Theta^!}{\vdash \langle \vec{\alpha}^!, \_ \rangle := \langle \vec{\tau}^a, \tau \rangle \rightsquigarrow \langle \vec{\tau}, \tau \rangle \mid \Theta^!}$$

$$\frac{\vdash \langle \vec{\alpha}^! \rangle := \langle \vec{\tau}^a \rangle \rightsquigarrow \langle \vec{\tau} \rangle \mid \Theta^!}{\vdash \langle \vec{\alpha}^!, \alpha \rangle := \langle \vec{\tau}^a, \mathbf{in}\ \tau_i\ \mathbf{out}\ \tau_o \rangle \rightsquigarrow \langle \vec{\tau}, \alpha \rangle \mid \Theta^!, \tau_i <: !\alpha <: \tau_o}$$

Subtyping $\Psi \mid \Theta \vdash \tau <: \tau \qquad \Psi \mid \Theta \vdash \tau^I <: \tau^I$

$$\frac{}{\Psi \mid \Theta \vdash \tau <: \tau} \qquad \frac{\Psi \mid \Theta \vdash \tau <: \tau' \qquad \Psi \mid \Theta \vdash \tau' <: \tau''}{\Psi \mid \Theta \vdash \tau <: \tau''}$$

$$\frac{}{\Psi \mid \Theta \vdash \bot <: \tau} \qquad \frac{}{\Psi \mid \Theta \vdash \tau <: \top}$$

$$\frac{\tau_\ell <: \nu\alpha <: \tau_u \in \Theta}{\Psi \mid \Theta \vdash \tau_\ell <: \alpha} \qquad \frac{\tau_\ell <: \nu\alpha <: \tau_u \in \Theta}{\Psi \mid \Theta \vdash \alpha <: \tau_u}$$

$$\frac{\Psi \mid \Theta \vdash \langle \vec{\tau}^a \rangle \rightsquigarrow \langle \vec{\tau} \rangle \mid \Theta^! \qquad \Psi \mid \Theta, \Theta^! \vdash I\langle \vec{\tau} \rangle \cap \tau <: \tau'}{\Psi \mid \Theta \vdash I\langle \vec{\tau}^a \rangle \cap \tau <: \tau'}$$

$$\frac{\begin{array}{c}\mathbf{interface}\ I\langle \Theta_I \rangle\ \mathbf{extends}\ \bullet\ \mathbf{satisfies}\ \bullet \in \Psi \\ \Psi \mid \Theta \vdash \langle \vec{\tau} \rangle <: \langle \vec{\tau}^a \rangle : \langle \Theta_I \rangle\end{array}}{\Psi \mid \Theta \vdash I\langle \vec{\tau} \rangle <: I\langle \vec{\tau}^a \rangle}$$

$$\frac{\mathbf{interface}\ I\langle \Theta_I \rangle\ \mathbf{extends}\ \tau^I\ \mathbf{satisfies}\ \bullet \in \Psi}{\Psi \mid \Theta \vdash I\langle \vec{\tau} \rangle <: \tau^I\lfloor \vec{\tau}/\Theta_I \rceil}$$

$$\frac{\mathbf{interface}\ I\langle \Theta_I \rangle\ \mathbf{extends}\ \ldots, \tau^I, \ldots\ \mathbf{satisfies}\ \bullet \in \Psi}{\Psi \mid \Theta \vdash I\langle \vec{\tau} \rangle <: \tau^I\lfloor \vec{\tau}/\Theta_I \rceil}$$

$$\frac{\begin{array}{c}\mathbf{interface}\ I\langle \Theta_I \rangle\ \mathbf{extends}\ \bullet\ \mathbf{satisfies}\ \bullet \in \Psi \\ \Psi \mid \Theta \vdash \langle \vec{\tau}_1 \rangle \cap \langle \vec{\tau}_2 \rangle <: \langle \vec{\tau}^a \rangle : \langle \Theta_I \rangle\end{array}}{\Psi \mid \Theta \vdash I\langle \vec{\tau}_1 \rangle \cap I\langle \vec{\tau}_2 \rangle <: I\langle \vec{\tau}^a \rangle}$$

$$\frac{i \in \{1,2\}}{\Psi \mid \Theta \vdash \tau_i <: \tau_1 \cup \tau_2} \qquad \frac{\Psi \mid \Theta \vdash \tau_1 <: \tau \qquad \Psi \mid \Theta \vdash \tau_2 <: \tau}{\Psi \mid \Theta \vdash \tau_1 \cup \tau_2 <: \tau}$$

$$\frac{\Psi \mid \Theta \vdash \tau <: \tau_1 \qquad \Psi \mid \Theta \vdash \tau <: \tau_2}{\Psi \mid \Theta \vdash \tau <: \tau_1 \cap \tau_2} \qquad \frac{i \in \{1,2\}}{\Psi \mid \Theta \vdash \tau_1 \cap \tau_2 <: \tau_i}$$

Figure 8.3 (contd.)

Type-Arguments Subtyping $\Psi \mid \Theta \vdash \langle \vec{\tau} \rangle <: \langle \vec{\tau}^a \rangle : \langle \Theta \rangle$

$$\frac{\Psi \mid \Theta \vdash \langle \tau_1 \rangle <: \langle \tau_1' \rangle : \langle \nu_1 \rangle \qquad \ldots}{\Psi \mid \Theta \vdash \langle \tau_1, \ldots \rangle <: \langle \tau_1', \ldots \rangle : \langle \bullet <: \nu_1 \alpha_1 <: \bullet, \ldots \rangle}$$

Type-Argument Subtyping $\Psi \mid \Theta \vdash \langle \tau \rangle <: \langle \tau^a \rangle : \langle \nu \rangle$

$$\frac{\Psi \mid \Theta \vdash \tau <: \tau'}{\Psi \mid \Theta \vdash \langle \tau \rangle <: \langle \tau' \rangle : \langle + \rangle} \qquad \frac{\Psi \mid \Theta \vdash \tau <: \tau' \qquad \Psi \mid \Theta \vdash \tau' <: \tau}{\Psi \mid \Theta \vdash \langle \tau \rangle <: \langle \tau' \rangle : \langle ! \rangle}$$

$$\frac{\Psi \mid \Theta \vdash \tau' <: \tau}{\Psi \mid \Theta \vdash \langle \tau \rangle <: \langle \tau' \rangle : \langle - \rangle} \qquad \frac{\Psi \mid \Theta \vdash \tau_i <: \tau \qquad \Psi \mid \Theta \vdash \tau <: \tau_u}{\Psi \mid \Theta \vdash \langle \tau \rangle <: \langle \mathbf{in}\ \tau_i\ \mathbf{out}\ \tau_o \rangle : \langle ! \rangle}$$

Intersected Type-Arguments Subtyping $\Psi \mid \Theta \vdash \langle \vec{\tau} \rangle \cap \langle \vec{\tau} \rangle <: \langle \vec{\tau}^a \rangle : \langle \Theta \rangle$

$$\frac{\Psi \mid \Theta \vdash \langle \tau_1 \rangle \cap \langle \tau_1' \rangle <: \langle \tau_1^a \rangle : \langle \nu_1 \rangle \qquad \ldots}{\Psi \mid \Theta \vdash \langle \tau_1, \ldots \rangle \cap \langle \tau_1', \ldots \rangle <: \langle \tau_1^a, \ldots \rangle : \langle \bullet <: \nu_1 \alpha_1 <: \bullet, \ldots \rangle}$$

Intersected Type-Argument Subtyping $\Psi \mid \Theta \vdash \langle \tau \rangle \cap \langle \tau \rangle <: \langle \tau^a \rangle : \langle \nu \rangle$

$$\frac{\Psi \mid \Theta \vdash \tau_1 \cap \tau_2 <: \tau}{\Psi \mid \Theta \vdash \langle \tau_1 \rangle \cap \langle \tau_2 \rangle <: \langle \tau \rangle : \langle + \rangle} \qquad \frac{\Psi \mid \Theta \vdash \tau <: \tau_1 \cup \tau_2}{\Psi \mid \Theta \vdash \langle \tau_1 \rangle \cap \langle \tau_2 \rangle <: \langle \tau \rangle : \langle - \rangle}$$

$$\frac{\Psi \mid \Theta \vdash \tau_i <: \tau_1 \cup \tau_2 \qquad \Psi \mid \Theta \vdash \tau_1 \cap \tau_2 <: \tau_o}{\Psi \mid \Theta \vdash \langle \tau_1 \rangle \cap \langle \tau_2 \rangle <: \langle \mathbf{in}\ \tau_i\ \mathbf{out}\ \tau_o \rangle : \langle ! \rangle}$$

Figure 8.3 (contd.)

$\vec{\tau}$	$\vec{\tau}\lfloor\vec{\tau}/\Theta\rceil$
$\tau_1, \ldots$	$\tau_1\lfloor\vec{\tau}/\Theta\rceil, \ldots$

τ	$\tau\lfloor\vec{\tau}/\Theta\rceil$
$\bot$	$\bot$
$\top$	$\top$
$I\langle\vec{\tau}^a\rangle$	$I\langle\vec{\tau}^a\lfloor\vec{\tau}/\Theta\rceil\rangle$
α_i	τ_i where $\Theta = \bullet <: \nu_1\alpha_1 <: \bullet, \ldots$ and $\vec{\tau} = \tau_1, \ldots$
α	α where $\nexists\tau_\ell, \nu, \tau_u.\ \tau_\ell <: \nu\alpha <: \tau_u \in \Theta$
$\tau_1 \cup \tau_2$	$\tau_1\lfloor\vec{\tau}/\Theta\rceil \cup \tau_2\lfloor\vec{\tau}/\Theta\rceil$
$\tau_1 \cap \tau_2$	$\tau_1\lfloor\vec{\tau}/\Theta\rceil \cap \tau_2\lfloor\vec{\tau}/\Theta\rceil$

$\vec{\tau}^a$	$\vec{\tau}^a\lfloor\vec{\tau}/\Theta\rceil$
$\tau_1^a, \ldots$	$\tau_1^a\lfloor\vec{\tau}/\Theta\rceil, \ldots$

τ^a	$\tau^a\lfloor\vec{\tau}/\Theta\rceil$
in τ_i **out** τ_o	**in** $\tau_i\lfloor\vec{\tau}/\Theta\rceil$ **out** $\tau_o\lfloor\vec{\tau}/\Theta\rceil$

τ^I	$\tau^I\lfloor\vec{\tau}/\Theta\rceil$
$I\langle\alpha_1, \ldots\rangle$	$I\langle\alpha_1\lfloor\vec{\tau}/\Theta\rceil, \ldots\rangle$
$I\langle\vec{\tau}\rangle$	$I\langle\vec{\tau}\lfloor\vec{\tau}/\Theta\rceil\rangle$

σ	$\sigma\lfloor\vec{\tau}/\Theta\rceil$
$S\langle\vec{\tau}\rangle$	$S\langle\vec{\tau}\lfloor\vec{\tau}/\Theta\rceil\rangle$

Σ	$\Sigma\lfloor\vec{\tau}/\Theta\rceil$ (belongs to Σ^τ)
$\varsigma_1 : \alpha_1.\sigma_1, \ldots$	$\varsigma_1 : \alpha_1\lfloor\vec{\tau}/\Theta\rceil.\sigma_1\lfloor\vec{\tau}/\Theta\rceil, \ldots$

Θ'	$\Theta'\lfloor\vec{\tau}/\Theta\rceil$
$\tau_1 <: \nu_1\alpha_1 <: \tau_1', \ldots$	$\tau_1\lfloor\vec{\tau}/\Theta\rceil <: \nu_1\alpha_1 <: \tau_1'\lfloor\vec{\tau}/\Theta\rceil, \ldots$

Γ	$\Gamma\lfloor\vec{\tau}/\Theta\rceil$
$p_1, \ldots$	$p_1\lfloor\vec{\tau}/\Theta\rceil, \ldots$

p	$p\lfloor\vec{\tau}/\Theta\rceil$
$x : \tau$	$x : \tau\lfloor\vec{\tau}/\Theta\rceil$
$f(\tau_1, \ldots) : \tau$	$f(\tau_1\lfloor\vec{\tau}/\Theta\rceil, \ldots) : \tau\lfloor\vec{\tau}/\Theta\rceil$

s	$s\lfloor\vec{\tau}/\Theta\rceil$ (belongs to s^τ)
$m\langle\Theta^!\rangle[\Theta'](\Gamma)[\Sigma] : \tau$	$m\langle\Theta^!\lfloor\vec{\tau}/\Theta\rceil\rangle[\Theta'\lfloor\vec{\tau}/\Theta\rceil](\Gamma\lfloor\vec{\tau}/\Theta\rceil)[\Sigma\lfloor\vec{\tau}/\Theta\rceil] : \tau\lfloor\vec{\tau}/\Theta\rceil$

Figure 8.4: Type-Argument Substitution

For $\text{Gen}_\cap$, joins and meets are trivially provided by arbitrary union and intersection types, the standard rules for which define them as the join and meet of the respective types. However, because union and intersection types introduce their own issues, as will be evident throughout the chapter, Gen_α instead ensures joins and meets exist by placing careful restrictions on the interface hierarchy.

First, suppose unrelated interfaces `B1` and `B2` were to both inherit unrelated interfaces `A1` and `A2`. Then the join $\texttt{B1} \sqcup \texttt{B2}$ needs to be a subtype of both `A1` and `A2`. In this case, this join would need to be the intersection type $\texttt{A1} \cap \texttt{A2}$. Thus, in order to avoid intersection types, Gen_α restricts its hierarchy to single inheritance.

Second, consider the case where we need to compute the join $I\langle \tau_+, \tau_-, \tau_! \rangle \sqcup I\langle \tau'_+, \tau'_-, \tau'_! \rangle$ in the hierarchy with interface $I\langle +\alpha_+, -\alpha_-, !\alpha_! \rangle$. For the covariant arguments, we can simply recursively compute their joins. For the contravariant arguments, we can recursively compute their *meets*. For the invariant arguments, we make use of *use-site variance* to mitigate differences between the type arguments, if any. That is, an $\texttt{Array}\langle \textbf{in } \tau_i \textbf{ out } \tau_o \rangle$ as an array of some unknown type but for which we can at least put in τ_i values and get out τ_o values. Since the **in** **out** construct effectively splits an invariant type into its contravariant and covariant uses, we can recursively compute the meet of the input type and the join of the output type. The input type must always be a subtype of the output type, but if they happen to even be equivalent then the subtyping rules in Figure 8.3 will ensure that using **in** τ_i **out** τ_o is equivalent to using simply τ_i or τ_o. Thus the join of this type in Gen_α should be $I\langle \tau_+ \sqcup \tau'_+, \tau_- \sqcap \tau'_-, \textbf{in } \tau_! \sqcap \tau'_! \textbf{ out } \tau_! \sqcup \tau'_! \rangle$.

At least we would like that to be the case. But recall that I might have constraints on its type parameters, and Gen_α needs those constraints

to be enforced in all corresponding type arguments. To make matters worse, if this constructed type fails to satisfy the constraints, that does not necessarily mean no common valid supertype exists. A valid supertype could be constructed by using less-precise type arguments that might happen to satisfy the constraints, *or* it could be constructed by upcasting to an interface that I inherits from that might impose more lax constraints or might not even reference the problematic type arguments at all. Worst yet, it is possible to construct valid supertypes using *both* of these strategies and with neither subsuming the other, in which case there simply is no join at all. This is why the rule for validating the bounds on type parameters in judgement $\Psi \vdash^{I} \langle\Theta\rangle$ of Figure 8.2 imposes variance requirements. These variance requirements ensure the property that, if some given type arguments for an interface I satisfy the required constraints, then all less-precise type-arguments (according to the variance of I) will also satisfy the required constraints. In particular this means that joining valid types using the strategy above will always result in a valid type. Conveniently, these variance requirements also ensure there is a principal way to remove a type variable from a valid type—say because the type variable is leaving scope—while retaining validity. This property is also necessary for type-checking $\text{Gen}_{\cap}$, which is why both calculi impose the variance requirements on interface type-parameter bounds.

Third, consider the case where we instead need to compute the meet $I\langle\tau_+, \tau_-, \tau_!\rangle \sqcap I\langle\tau'_+, \tau'_-, \tau'_!\rangle$ in the same hierarchy. The construction is as before but with recursive joins and meets swapped:

$$I\langle\tau_+ \sqcap \tau'_+, \tau_- \sqcup \tau'_-, \textbf{in } \tau_! \sqcup \tau'_! \textbf{ out } \tau_! \sqcap \tau'_!\rangle$$

However, in this case the design of $\Psi \vdash^I \langle\Theta\rangle$ does not ensure this type is valid. Instead, it ensures that, if this type is *invalid*, then no more-precise instantiation of I is valid either. Thus, after constructing this type, one simply checks the constraints and, if any of them fails, determines that $\bot$ is necessarily the meet of the two types instead.

Lastly, consider the case where one type is an instantiation of I and the other type is an instantiation of J, which I inherits from. When joining the two types, one simply upcasts the instantiation of I into an instantiation of J and proceeds appropriately. But when meeting the two types, one needs to figure how to construct an instantiation of I that is a subtype of both types. That is, one needs to figure out how to incorporate the type arguments to J appropriately into type arguments for I. In general, this is impossible, such as if $I\langle\alpha\rangle$ inherits $J\langle\top\rangle$, in which case no meet exists at all. This is why Gen_α restricts the grammar of inherited interfaces τ^I so that only type parameters can be used as type arguments. This restriction makes it clear how to incorporate the type arguments to J appropriately into type arguments for I, ensuring the existence of meets.

8.3.5 *Type-Argument Inference*

$\text{Gen}_\cap$ bypasses all the issues with joins and meets by providing arbitrary union and intersection types, but as we suggested these types introduce their own issues. One such issue arises in type-argument inference. Consider the following generic method:

```
flatten⟨E⟩(seq_of_seqs : Seq⟨Seq⟨E⟩⟩) : Seq⟨E⟩
```

This common method seems like it should always be principally inferable, but naïvely adding arbitrary intersection types makes it not so.

Suppose we call `flatten` on a value of type $\texttt{Seq}\langle\texttt{Seq}\langle\texttt{Int}\rangle \cap \texttt{Seq}\langle\texttt{String}\rangle\rangle$. Then `E` could be inferred to be either `Int` or `String`, with neither being better than the other. In fact, instead of intersection types one can use multiple-instantiation inheritance to create a similar ambiguity in C#, one that is even semantically meaningful.

For this reason, $\text{Gen}_\cap$ enforces *principal*-instantiation inheritance in its interface hierarchy. That is, the design of the judgement $\Psi \mid \Psi \mid \Theta \vdash^I \vec{\tau}^I$ in Figure 8.2 ensures that if an interface inherits both $\texttt{Seq}\langle\texttt{Int}\rangle$ and $\texttt{Seq}\langle\texttt{String}\rangle$ through its various inherited interfaces, then it must also inherit some principal instantiation of `Seq`, which would likely be $\texttt{Seq}\langle\bot\rangle$ in this case due to the covariance of `Seq` and the presumable disjointness of `Int` and `String`.

$\text{Gen}_\cap$ then incorporates this invariant into its subtyping system through the second $\text{Gen}_\cap$ subtyping rule in Figure 8.3. This rule allows the type arguments of intersections of interface types to be combined according to their respective variances, as done by judgement $\Psi \mid \Theta \vdash \langle\vec{\tau}\rangle \cap \langle\vec{\tau}\rangle <: \langle\vec{\tau}^a\rangle : \langle\Theta\rangle$. In particular, $\texttt{Seq}\langle\texttt{Int}\rangle \cap \texttt{Seq}\langle\texttt{String}\rangle$ is considered to be a subtype of $\texttt{Seq}\langle\texttt{Int} \cap \texttt{String}\rangle$ due to covariance of `Seq`. This extension to subtyping, which we know to be decidable based on the results in Chapter 3, makes the type argument for `flatten` always inferable. In the concrete case above, `E` would be inferred to be $\texttt{Int} \cap \texttt{String}$.

8.4 SHAPES AND SATISFACTION

With the basics and invariants of types and subtyping in place, we now discuss our solution to the binary-method problem: shapes. The second pass in validating a hierarchy ensures that shapes and subshaping are well-behaved. The third pass ensures that shape satisfaction is well-behaved. Note that, unlike with interfaces, the design of the two calculi are nearly identical with respect to shapes.

8.4.1 *Shapes*

A shape declaration c^S, as shown in Figure 8.5, specifies constrained type parameters Θ, inherited shapes σ, and signatures s of methods guaranteed by the shape. In the second pass, we validate shapes and the inheritance hierarchy between shapes, ignoring interface declarations entirely.

In validating a shape declaration, validation of the constrained type parameters is much like as with interfaces. The only difference is that here neither calculus imposes any variance requirements on the bounds. This is because shapes are not involved in types and so are not intermediately constructed through joins and meets. For this same reason, shapes do not have **in out** type arguments. However, shapes are used in type-argument inference, so we require principal-instantiation inheritance between shapes. This is all that comprises the second pass of hierarchy validation.

Thus a shape is very similar to an interface. Much like how a type being a subtype of an interface ensures it has certain methods, a type satisfying a shape also ensures it has certain methods. But, as discussed

$$\text{Shape Name } S \qquad \text{Shape } \sigma \ ::= \ S\langle\vec{\tau}\rangle$$
$$\text{Shape Declaration } c^S ::= \ \textbf{shape}\ S\langle\Theta\rangle\ \textbf{extends}\ \sigma, \ldots\ \{s; \ldots\}$$

Shapes Validity $\Psi \vdash^S \Psi$

$$\frac{}{\Psi \vdash^S \varnothing} \qquad \frac{\Psi \vdash^S \Psi'}{\Psi \mid \Psi' \vdash^S \Psi'; c^I} \qquad \frac{\Psi \vdash^S \Psi' \qquad \Psi \vdash^S \langle\Theta\rangle \qquad \Psi \mid \Psi' \mid \Theta \vdash^S \sigma_1, \ldots}{\Psi \vdash^S \Psi'; \textbf{shape}\ S\langle\Theta\rangle\ \textbf{extends}\ \sigma_1, \ldots\ \{\bullet\}}$$

Shape Parameter Validity $\Psi \vdash^S \langle\Theta\rangle$

$$\frac{}{\Psi \vdash^S \langle\rangle} \qquad \frac{\Psi \vdash^S \langle\Theta\rangle \qquad \Psi \mid \Theta \vdash_? \tau_\ell \qquad \Psi \mid \Theta \vdash_? \tau_u \qquad \Psi \mid \Theta \vdash \tau_\ell <: \tau_u}{\Psi \vdash^S \langle\Theta, \tau_\ell <: \nu\alpha <: \tau_u\rangle}$$

Shape Inheritance Validity $\Psi \mid \Psi \mid \Theta \vdash^S \sigma, \ldots$

$$\frac{}{\Psi \mid \Psi' \mid \Theta \vdash^S \varnothing}$$

$$\frac{\begin{array}{c}\Psi \mid \Psi_S \mid \Theta \vdash^S \sigma_1, \ldots \\ \Psi \mid \Theta \vdash_+ \langle\vec{\tau}\rangle : \langle\Theta_S\rangle \qquad \Psi \mid \Theta \vdash \sigma_1, \ldots \sqsubset: \sigma_1'\lfloor\vec{\tau}/\Theta_S\rceil \qquad \ldots\end{array}}{\Psi \mid \Psi_S; \textbf{shape}\ S\langle\Theta_S\rangle\ \textbf{extends}\ \sigma_1', \ldots\ \{\bullet\}; \ldots \mid \Theta \vdash^S \sigma_1, \ldots, S\langle\vec{\tau}\rangle}$$

Shape Covering $\Psi \mid \Theta \vdash^S \sigma, \ldots \sqsubseteq \sigma$

$$\frac{\Psi \mid \Theta \vdash \sigma \sqsubset: \sigma'}{\Psi \mid \Theta \vdash^S \ldots, \sigma, \ldots \sqsubseteq \sigma'}$$

Shape Validity and Variance $\Psi \mid \Theta \vdash_{\nu^?} \sigma$

$$\frac{\textbf{shape}\ S\langle\Theta_S\rangle\ \textbf{extends}\ \bullet\ \{\bullet\} \in \Psi \qquad \Psi \mid \Theta \vdash_{\nu^?} \langle\vec{\tau}\rangle : \langle\Theta_S\rangle}{\Psi \mid \Theta \vdash_{\nu^?} S\langle\vec{\tau}\rangle}$$

Subshaping $\Psi \mid \Theta \vdash \sigma \sqsubset: \sigma$

$$\frac{}{\Psi \mid \Theta \vdash \sigma \sqsubset: \sigma} \qquad \frac{\Psi \mid \Theta \vdash \sigma \sqsubset: \sigma' \qquad \Psi \mid \Theta \vdash \sigma' \sqsubset: \sigma''}{\Psi \mid \Theta \vdash \sigma \sqsubset: \sigma''}$$

$$\frac{\begin{array}{c}\textbf{shape}\ S\langle\Theta_S\rangle\ \textbf{extends}\ \bullet\ \{\bullet\} \in \Psi \\ \Psi \mid \Theta \vdash \langle\vec{\tau}\rangle <: \langle\vec{\tau}'\rangle : \langle\Theta_S\rangle\end{array}}{\Psi \mid \Theta \vdash S\langle\vec{\tau}\rangle \sqsubset: S\langle\vec{\tau}'\rangle} \qquad \frac{\textbf{shape}\ S\langle\Theta_S\rangle\ \textbf{extends}\ \ldots, \sigma, \ldots\ \{\bullet\} \in \Psi}{\Psi \mid \Theta \vdash S\langle\vec{\tau}\rangle \sqsubset: \sigma\lfloor\vec{\tau}/\Theta_S\rceil}$$

Figure 8.5: Shapes

in Chapter 2, how they are used in the design of libraries and in the type system are very different. That is, we are simply incorporating the programmer behavior observed by material-shape separation directly into the syntax of the design of the language in order to ensure decidability and provide new functionality, as discussed next.

8.4.2 *Shape Satisfaction*

In the third pass, we validate conditionally satisfied shapes of interfaces in order to ensure that shape satisfaction is well-behaved. Every interface declares a list of conditionally satisfied shapes σ^I. As shown in Figure 8.6, a conditionally satisfied shape is a shape σ along with type parameters Θ and a shape context Σ. The idea is that the shape context specifies what shapes must be satisfied by the various type parameters of the interface and Θ in order for the interface to itself satisfy σ. For example, the interface $\texttt{Seq}\langle+\alpha\rangle$ could specify that it satisfies $\texttt{Eq}\langle\texttt{Seq}\langle\beta\rangle\rangle$ provided that β is a supertype of α and α satisfies $\texttt{Eq}\langle\beta\rangle$. This would be syntactically written as **interface** $\texttt{Seq}\langle+\alpha\rangle$ **satisfies** $\texttt{Eq}\langle\texttt{Seq}\langle\beta\rangle\rangle[\alpha <: -\beta][\varsigma : \alpha.\texttt{Eq}\langle\beta\rangle]$ (where the evidence variables ς are insignificant throughout this section), and this definition is in fact covariant with respect to α.

In validating conditionally satsified shapes, conceptually the first thing we check is that each conditionally satisfied shape is covariant with respect to the type parameters of the interface. This ensures that the more precise the type arguments are then the easier it will be to satisfy the conditions and the more precise the resulting shape will be, both of which are important for decidability. Consequently, only covariant and invariant type

$$
\begin{array}{lll}
\text{Evidence Variable} & \varsigma & \\
\text{Conditionally Satisfied Shapes} & \vec{\sigma}^I & ::= \sigma^I, \ldots \\
\text{Conditionally Satisfied Shape} & \sigma^I & ::= \sigma[\Theta][\Sigma] \\
\text{Shape Context} & \Sigma & ::= \varsigma : \alpha.\sigma, \ldots \\
\text{Shape Premise} & \Sigma^\tau & ::= \varsigma : \tau.\sigma, \ldots \\
\text{Shape Conclusion} & \Sigma^{<:} & ::= \bot \mid \langle\Theta\rangle[\Sigma]
\end{array}
$$

Shape Satisfactions Validity $\Psi \vdash^\sigma \Psi$

$$\frac{}{\Psi \vdash^\sigma \varnothing} \qquad \frac{\Psi \vdash^\sigma \Psi'}{\Psi \vdash^\sigma \Psi'; c^S} \qquad \frac{\begin{array}{c}\Psi \vdash^\sigma \Psi' \qquad \Psi \mid \Psi' \mid \Psi' \mid \Theta \vdash^\sigma \vec{\sigma}^I \\ \Psi \mid \Theta \vdash^\sigma \vec{\sigma}^I \sqsubseteq \tau_1^I \qquad \ldots\end{array}}{\Psi \vdash^\sigma \Psi'; \mathbf{interface}\ I\langle\Theta\rangle\ \mathbf{extends}\ \tau_1^I, \ldots\ \mathbf{satisfies}\ \vec{\sigma}^I\ \{\bullet\}}$$

Conditionally Satisfied Shapes Validity $\Psi \mid \Psi \mid \Psi \mid \Theta \vdash^\sigma \vec{\sigma}^I$

$$\frac{}{\Psi \mid \Psi' \mid \Theta \vdash^\sigma \varnothing} \qquad \frac{\begin{array}{c}\Psi \mid \Psi_I \mid \Psi_S \mid \Theta \vdash^\sigma \vec{\sigma}^I \\ \Psi_I \mid \Theta \vdash \langle\Theta'\rangle \qquad \Psi \mid \Theta, \Theta' \vdash_+ \langle\vec{\tau}\rangle : \langle\Theta_S\rangle \qquad \Psi \mid \Theta, \Theta' \mid \varnothing \vdash [\Sigma] \\ \Psi \mid \Theta \vdash ()[\Sigma] \rightsquigarrow \langle\Theta' \mid \varnothing\rangle \qquad \Psi \mid \Theta, \Theta' \mid \Sigma \vdash^\sigma \vec{\sigma}^I \sqsubseteq \sigma_1\lfloor\vec{\tau}/\Theta_S\rceil \qquad \ldots\end{array}}{\Psi \mid \Psi_I \mid \Psi_S; \mathbf{shape}\ S\langle\Theta_S\rangle\ \mathbf{extends}\ \sigma_1, \ldots\ \{\bullet\}; \ldots \mid \Theta \vdash^\sigma \vec{\sigma}^I, S\langle\vec{\tau}\rangle[\Theta'][\Sigma]}$$

Conditionally Satisfied Shape Covering $\Psi \mid \Theta \mid \Sigma^\tau \vdash^\sigma \vec{\sigma}^I \sqsubseteq \sigma$

$$\frac{\Psi \mid \Theta \mid \Sigma^\tau \mid \varnothing \vdash \bot}{\Psi \mid \Theta \mid \Sigma^\tau \vdash^\sigma \vec{\sigma}^I \sqsubseteq \sigma'} \qquad \frac{\begin{array}{c}\Psi \mid \Theta \mid \Sigma^\tau \mid \varnothing \vdash \langle\Theta'\rangle[\Sigma'] \qquad \Psi \mid \Theta' \vdash_? \langle\vec{\tau}\rangle : \langle\Theta_\sigma\rangle \\ \Psi \mid \Theta' \mid \Sigma' \vdash \Sigma_\sigma\lfloor\vec{\tau}/\Theta_\sigma\rceil \qquad \Psi \mid \Theta' \vdash \sigma\lfloor\vec{\tau}/\Theta_\sigma\rceil \sqsubset: \sigma'\end{array}}{\Psi \mid \Theta \mid \Sigma^\tau \vdash^\sigma \ldots, \sigma[\Theta_\sigma][\Sigma_\sigma], \ldots \sqsubseteq \sigma'}$$

Conditionally Satisfied Shape Inheritance $\Psi \mid \Theta \vdash^\sigma \vec{\sigma}^I \sqsubseteq \tau^I$

$$\frac{\begin{array}{c}\mathbf{interface}\ I\langle\Theta_I\rangle\ \mathbf{extends}\ \bullet\ \mathbf{satisfies}\ \sigma_1[\Theta_1][\Sigma_1], \ldots\ \{\bullet\} \in \Psi \\ \Psi \mid \Theta, \Theta_1 \mid \Sigma_1\lfloor\vec{\tau}/\Theta_I\rceil \vdash^\sigma \vec{\sigma}^I \sqsubseteq \sigma_1 \qquad \ldots\end{array}}{\Psi \mid \Theta \vdash^\sigma \vec{\sigma}^I \sqsubseteq I\langle\vec{\tau}\rangle}$$

Shape-Context Validity $\Psi \mid \Theta \mid \Sigma \vdash [\Sigma]$

$$\frac{}{\Psi \mid \Theta \mid \Sigma \vdash []} \qquad \frac{\begin{array}{c}\Psi \mid \Theta \mid \Sigma \vdash [\Sigma'] \qquad \mathbf{shape}\ S\langle\Theta_S\rangle\ \mathbf{extends}\ \sigma_1, \ldots\ \{\bullet\} \in \Psi \\ \tau_\ell <: \nu\alpha <: \tau_u \in \Theta \qquad \vdash \nu \leq + \qquad \Psi \mid \Theta \vdash_- \langle\vec{\tau}_\alpha\rangle : \langle\Theta_S\rangle \\ \Psi \mid \Theta \mid \Sigma, \Sigma' \vdash \tau_\ell.\sigma \qquad \Psi \mid \Theta \mid \Sigma, \Sigma' \vdash \alpha.\sigma_1\lfloor\vec{\tau}/\Theta_S\rceil \qquad \ldots \\ \forall\vec{\tau}.\ \Psi \mid \Theta \mid \Sigma, \Sigma' \vdash \tau_u.S\langle\vec{\tau}\rangle \implies \Psi \mid \Theta \vdash \langle\vec{\tau}_\alpha\rangle <: \langle\vec{\tau}\rangle : \langle\Theta_S\rangle\end{array}}{\Psi \mid \Theta \mid \Sigma \vdash [\Sigma', \varsigma : \alpha.S\langle\vec{\tau}_\alpha\rangle]}$$

Figure 8.6: Conditionally Satisfied Shapes (see also Figures 8.7, 8.8)

parameters can be constrained. We also check that, for each conditionally satisfied shape, the type parameters in Θ can be decidably and principally inferred from how they are used in the conditioning shape context, which is conceptually a special case of the type-argument inference of methods that we will discuss later.

The second thing we check is that every shape has a principal conditionally satisfied shape (if any) amongst the list. Similarly, the final thing we check is that the conditional shapes declared by the interface subsume those declared by the interfaces it inherits. These are important for type-argument inference, and as a formalization convenience our particular means of formulating these requirements ensure that all shapes conditionally satisfied by the interface, including those that would be indirectly satisfied via shape or interface inheritance, are necessarily in the list.

8.4.3 *Shape Simplification*

Both of these latter checks are done using the judgement $\Psi \mid \Theta \mid \Sigma^\tau \vdash^\sigma \vec{\sigma}^I \sqsubseteq \sigma$. This judgement ensures that, given evidence that the shapes in Σ^τ are satisfiable, then the methods guaranteed by σ are guaranteed by conditionally satisfied shapes in $\vec{\sigma}^I$. This judgement has two rules, both of which use the judgement $\Psi \mid \Theta \mid \Sigma^\tau \mid \varnothing \vdash \Sigma^{<:}$, where $\Sigma^{<:}$ is a conclusion that can be deduced from the premise. This conclusion can either be that the premise unsatisfiable and therefore the relevant method can never be invoked (denoted $\bot$), or that some stronger subtyping constraints and shape evidence necessarily hold (denoted $\langle\Theta\rangle[\Sigma]$).

$$\text{Inferred Bounds } \Theta^{<:} ::= \varnothing \mid \Theta^{<:}, \tau <: \alpha \mid \Theta^{<:}, \alpha <: \tau$$

Shape Simplification $\Psi \mid \Theta \mid \Sigma^{\tau} \mid \Theta^{<:} \vdash \Sigma^{<:}$

$$\frac{\Psi \mid \Theta^{<:} \vdash \langle \Theta \rangle \rightsquigarrow \langle \Theta' \rangle \qquad \Psi \mid \Theta' \mid \Sigma \vdash \Sigma'}{\Psi \mid \Theta \mid \Sigma \mid \Theta^{<:} \vdash \langle \Theta' \rangle [\Sigma']}$$

$$\frac{\begin{array}{c}\textbf{interface } I\langle \Theta_I \rangle \textbf{ extends } \bullet \textbf{ satisfies } \ldots, S\langle \vec{\tau}'_S \rangle [\Theta'][\Sigma'], \ldots \{\bullet\} \in \Psi \\ \Psi \mid \Theta \vdash \langle \vec{\tau}^a_I \rangle \rightsquigarrow \langle \vec{\tau}_I \rangle \mid \Theta^! \qquad \Psi \mid \varnothing \vdash S\langle \vec{\tau}'_S \lfloor \vec{\tau}_I / \Theta_I \rceil \rangle \sqsubset: S\langle \vec{\tau}_S \rangle \rightsquigarrow \langle \Theta^{<:}_S \rangle \\ \Psi \mid \Theta, \Theta^!, \Theta' \mid \Sigma^{\tau}_1, \Sigma' \lfloor \vec{\tau}_I / \Theta_I \rceil, \Sigma^{\tau}_2 \mid \Theta^{<:}, \Theta^{<:}_S \vdash \Sigma^{<:}\end{array}}{\Psi \mid \Theta \mid \Sigma^{\tau}_1, \varsigma : I\langle \vec{\tau}^a_I \rangle . S\langle \vec{\tau}_S \rangle, \Sigma^{\tau}_2 \mid \Theta^{<:} \vdash \Sigma^{<:}}$$

$$\frac{\begin{array}{c}\textbf{interface } I\langle \bullet \rangle \textbf{ extends } \bullet \textbf{ satisfies } \sigma^I_1, \ldots \{\bullet\} \in \Psi \\ \nexists \vec{\tau}', \Theta' \Sigma' .\ S\langle \vec{\tau}' \rangle [\Theta'][\Sigma'] \in \{\sigma^I_1, \ldots\}\end{array}}{\Psi \mid \Theta \mid \ldots, \varsigma : I\langle \bullet \rangle . S\langle \bullet \rangle, \ldots \mid \Theta^{<:} \vdash \bot}$$

$$\frac{\tau_\ell <: \nu\alpha <: \tau_u \in \Theta \qquad \Psi \mid \Theta \mid \Sigma^{\tau}_1, \varsigma : \alpha.\sigma, \varsigma : \tau_\ell.\sigma, \Sigma^{\tau}_2 \mid \Theta^{<:} \vdash \Sigma^{<:}}{\Psi \mid \Theta \mid \Sigma^{\tau}_1, \varsigma : \alpha.\sigma, \Sigma^{\tau}_2 \mid \Theta^{<:} \vdash \Sigma^{<:}}$$

$$\frac{}{\Psi \mid \Theta \mid \ldots, \varsigma : \top.\sigma, \ldots \mid \Theta^{<:} \vdash \bot} \qquad \frac{\Psi \mid \Theta \mid \Sigma^{\tau}_1, \Sigma^{\tau}_2 \mid \Theta^{<:} \vdash \Sigma^{<:}}{\Psi \mid \Theta \mid \Sigma^{\tau}_1, \varsigma : \bot.\sigma, \Sigma^{\tau}_2 \mid \Theta^{<:} \vdash \Sigma^{<:}}$$

$$\frac{\begin{array}{c}\Psi \mid \Theta \mid \Sigma^{\tau}_1, \tau_1.\sigma, \Sigma^{\tau}_2 \mid \Theta^{<:} \vdash \Sigma^{<:} \\ \Psi \mid \Theta \mid \Sigma^{\tau}_1, \tau_2.\sigma, \Sigma^{\tau}_2 \mid \Theta^{<:} \vdash \Sigma^{<:}\end{array}}{\Psi \mid \Theta \mid \Sigma^{\tau}_1, (\tau_1 \cap \tau_2).\sigma, \Sigma^{\tau}_2 \mid \Theta^{<:} \vdash \Sigma^{<:}} \qquad \frac{\Psi \mid \Theta \mid \Sigma^{\tau}_1, \tau_1.\sigma, \tau_2.\sigma, \Sigma^{\tau}_2 \mid \Theta^{<:} \vdash \Sigma^{<:}}{\Psi \mid \Theta \mid \Sigma^{\tau}_1, (\tau_1 \cup \tau_2).\sigma, \Sigma^{\tau}_2 \mid \Theta^{<:} \vdash \Sigma^{<:}}$$

Shape Satisfaction $\Psi \mid \Theta \mid \Sigma \vdash \Sigma^{\tau} \qquad \Psi \mid \Theta \mid \Sigma \vdash \tau.\sigma$

$$\frac{\Psi \mid \Theta \mid \Sigma \vdash \tau_1.\sigma_1 \qquad \ldots}{\Psi \mid \Theta \mid \Sigma \vdash \varsigma_1 : \tau_1.\sigma_1, \ldots} \qquad \frac{\begin{array}{c}\Psi \mid \Theta \mid \Sigma \vdash \tau.\sigma \\ \Psi \mid \Theta \vdash \tau' <: \tau \qquad \Psi \mid \Theta \vdash \sigma \sqsubset: \sigma'\end{array}}{\Psi \mid \Theta \mid \Sigma \vdash \tau'.\sigma'}$$

$$\frac{\varsigma : \alpha.\sigma \in \Sigma}{\Psi \mid \Theta \mid \Sigma \vdash \alpha.\sigma} \qquad \frac{}{\Psi \mid \Theta \mid \Sigma \vdash \bot.\sigma} \qquad \frac{\Psi \mid \Theta \mid \Sigma \vdash \tau_1.\sigma \qquad \Psi \mid \Theta \mid \Sigma \vdash \tau_2.\sigma}{\Psi \mid \Theta \mid \Sigma \vdash (\tau_1 \cup \tau_2).\sigma}$$

$$\frac{\begin{array}{c}\textbf{interface } I\langle \Theta_I \rangle \textbf{ extends } \bullet \textbf{ satisfies } \ldots, \sigma'[\Theta'][\Sigma'], \ldots \{\bullet\} \in \Psi \\ \Psi \mid \Theta \vdash \langle \vec{\tau}^a \rangle \rightsquigarrow \langle \vec{\tau} \rangle \mid \Theta^! \qquad \Psi \mid \Theta, \Theta^! \vdash_? \langle \vec{\tau}' \rangle : \langle \Theta' \lfloor \vec{\tau} / \Theta_I \rceil \rangle \\ \Psi \mid \Theta, \Theta^! \mid \Sigma \vdash \Sigma' \lfloor \vec{\tau}, \vec{\tau}' / \Theta_I, \Theta' \rceil \qquad \Psi \mid \Theta, \Theta^! \vdash \sigma' \lfloor \vec{\tau}, \vec{\tau}' / \Theta_I, \Theta' \rceil \sqsubset: \sigma\end{array}}{\Psi \mid \Theta \mid \Sigma \vdash I\langle \vec{\tau}^a \rangle . \sigma}$$

$$\frac{\begin{array}{c}\textbf{shape } S\langle \Theta_S \rangle \textbf{ extends } \bullet \ \{\bullet\} \in \Psi \qquad \Psi \mid \Theta \mid \Sigma \vdash \tau.S\langle \vec{\tau}_1 \rangle \\ \Psi \mid \Theta \mid \Sigma \vdash \tau.S\langle \vec{\tau}_2 \rangle \qquad \Psi \mid \Theta \vdash \langle \vec{\tau}_1 \rangle \cap \langle \vec{\tau}_2 \rangle <: \langle \vec{\tau} \rangle : \langle \Theta_S \rangle\end{array}}{\Psi \mid \Theta \mid \Sigma \vdash \tau.S\langle \vec{\tau} \rangle}$$

Figure 8.7: Shape Satisfaction

Invariant Conversion $\vdash \langle\Theta\rangle \rightsquigarrow \langle\Theta^!\rangle$

$$\frac{}{\vdash \langle\rangle \rightsquigarrow \langle\rangle} \qquad \frac{\vdash \langle\Theta\rangle \rightsquigarrow \langle\Theta^!\rangle}{\vdash \langle\Theta, \tau_\ell <: \nu\alpha <: \tau_u\rangle \rightsquigarrow \langle\Theta^!, \tau_\ell <: !\alpha <: \tau_u\rangle}$$

Kind-Context Validity $\Psi \mid \Theta \vdash \langle\Theta\rangle$

$$\frac{}{\Psi \mid \Theta \vdash \langle\rangle} \qquad \frac{\Psi \mid \Theta \vdash_+ \tau_\ell \qquad \Psi \mid \Theta \vdash_- \tau_u \qquad \Psi \mid \Theta \vdash \tau_\ell <: \tau_u \qquad \Psi \mid \Theta, \tau_\ell <: !\alpha <: \tau_u \vdash \langle\Theta'\rangle}{\Psi \mid \Theta \vdash \langle\tau_\ell <: \nu\alpha <: \tau_u, \Theta'\rangle}$$

Inferred-Bound Incorporation $\Psi \mid \Theta^{<:} \vdash \langle\Theta\rangle \rightsquigarrow \langle\Theta'\rangle$

$$\frac{}{\Psi \mid \Theta^{<:} \vdash \langle\Theta\rangle \rightsquigarrow \langle\Theta\rangle}$$

$$\frac{\Psi \mid \Theta^{<:} \vdash \langle\Theta\rangle \rightsquigarrow \langle\Theta_1, \tau_\ell <: \nu\alpha <: \tau_u, \Theta_2\rangle \quad \tau'_\ell <: \alpha \in \Theta^{<:} \qquad \Psi \mid \Theta_1 \vdash_? \tau''_\ell \quad \Psi \mid \Theta_1 \vdash \tau_\ell <: \tau''_\ell \qquad \Psi \mid \Theta \vdash \tau''_\ell <: \tau'_\ell \qquad \Psi \mid \Theta_1 \vdash \tau''_\ell <: \tau_u}{\Psi \mid \Theta^{<:} \vdash \langle\Theta\rangle \rightsquigarrow \langle\Theta_1, \tau''_\ell <: \nu\alpha <: \tau_u, \Theta_2\rangle}$$

$$\frac{\Psi \mid \Theta^{<:} \vdash \langle\Theta\rangle \rightsquigarrow \langle\Theta_1, \tau_\ell <: \nu\alpha <: \tau_u, \Theta_2\rangle \quad \alpha <: \tau'_u \in \Theta^{<:} \qquad \Psi \mid \Theta_1 \vdash_? \tau''_u \quad \Psi \mid \Theta_1 \vdash \tau_\ell <: \tau''_u \qquad \Psi \mid \Theta \vdash \tau'_u <: \tau''_u \qquad \Psi \mid \Theta_1 \vdash \tau''_u <: \tau_u}{\Psi \mid \Theta^{<:} \vdash \langle\Theta\rangle \rightsquigarrow \langle\Theta_1, \tau_\ell <: \nu\alpha <: \tau''_u, \Theta_2\rangle}$$

Type Bound Inference $\Psi \mid \Theta \vdash \tau <: \tau \rightsquigarrow \langle\Theta^{<:}\rangle$

$$\frac{}{\Psi \mid \Theta \vdash \tau <: \alpha \rightsquigarrow \langle\tau <: \alpha\rangle} \qquad \frac{\Psi \mid \Theta \vdash_I \tau <: \langle\vec{\tau}^a\rangle \rightsquigarrow \langle\Theta^{<:}\rangle}{\Psi \mid \Theta \vdash \tau <: I\langle\vec{\tau}^a\rangle \rightsquigarrow \langle\Theta^{<:}\rangle}$$

$$\frac{}{\Psi \mid \Theta \vdash \alpha <: \tau \rightsquigarrow \langle\alpha <: \tau\rangle} \qquad \frac{\Psi \mid \Theta \vdash \tau <: \tau_1 \rightsquigarrow \langle\Theta_1^{<:}\rangle \quad \Psi \mid \Theta \vdash \tau <: \tau_2 \rightsquigarrow \langle\Theta_2^{<:}\rangle}{\Psi \mid \Theta \vdash \tau <: \tau_1 \cap \tau_2 \rightsquigarrow \langle\Theta_1^{<:}, \Theta_2^{<:}\rangle}$$

$$\frac{}{\Psi \mid \Theta \vdash \tau <: \tau' \rightsquigarrow \langle\rangle}$$

Shape Bound Inference $\Psi \mid \Theta \vdash \sigma \sqsubset: \sigma \rightsquigarrow \langle\Theta^{<:}\rangle$

$$\frac{\textbf{shape}\ S\langle\Theta_S\rangle\ \textbf{extends}\ \bullet\ \{\bullet\} \in \Psi \qquad \Psi \mid \Theta \vdash \langle\vec{\tau}\rangle <: \langle\vec{\tau}'\rangle : \langle\Theta_S\rangle \rightsquigarrow \langle\Theta^{<:}\rangle}{\Psi \mid \Theta \vdash S\langle\vec{\tau}\rangle \sqsubset: S\langle\vec{\tau}'\rangle \rightsquigarrow \langle\Theta^{<:}\rangle}$$

$$\frac{\textbf{shape}\ S_1\langle\Theta_1\rangle\ \textbf{extends}\ \ldots, S_2\langle\vec{\tau}\rangle, \ldots\ \{\bullet\} \in \Psi \quad \textbf{shape}\ S_2\langle\Theta_2\rangle\ \textbf{extends}\ \bullet\ \{\bullet\} \in \Psi \qquad \Psi \mid \Theta \vdash \langle\vec{\tau}\lfloor^{\vec{\tau}_1}/_{\Theta_1}\rceil\rangle <: \langle\vec{\tau}_2\rangle : \langle\Theta_2\rangle \rightsquigarrow \langle\Theta^{<:}\rangle}{\Psi \mid \Theta \vdash S_1\langle\vec{\tau}_1\rangle \sqsubset: S_2\langle\vec{\tau}_2\rangle \rightsquigarrow \langle\Theta^{<:}\rangle}$$

Figure 8.7 (contd.)

Intersected Interface Bound Inference $\Psi \mid \Theta \vdash_I \tau, \ldots <: \langle \vec{\tau}^a \rangle \rightsquigarrow \langle \Theta^{<:} \rangle$

$$\frac{\Psi \mid \Theta \vdash \langle \vec{\tau}^a \rangle \rightsquigarrow \langle \vec{\tau} \rangle \mid \Theta^! \qquad \Psi \mid \Theta, \Theta^! \vdash_I \vec{\tau}_1, I'\langle \vec{\tau} \rangle, \vec{\tau}_2 <: \langle \vec{\tau}_I^a \rangle \rightsquigarrow \langle \Theta^{<:} \rangle}{\Psi \mid \Theta \vdash_I \vec{\tau}_1, I'\langle \vec{\tau}^a \rangle, \vec{\tau}_2 <: \langle \vec{\tau}_I^a \rangle \rightsquigarrow \langle \Theta^{<:} \rangle}$$

$$\frac{\textbf{interface } I\langle \Theta_I \rangle \textbf{ extends } \bullet \textbf{ satisfies } \bullet \ \{\bullet\} \in \Psi \qquad \Psi \mid \Theta \vdash \langle \vec{\tau}_1 \rangle, \ldots <: \langle \vec{\tau}_I^a \rangle : \langle \Theta_I \rangle \rightsquigarrow \langle \Theta^{<:} \rangle}{\Psi \mid \Theta \vdash_I I\langle \vec{\tau}_1 \rangle, \ldots <: \langle \vec{\tau}_I^a \rangle \rightsquigarrow \langle \Theta^{<:} \rangle}$$

$$\frac{\textbf{interface } I'\langle \Theta' \rangle \textbf{ extends } \ldots, I\langle \vec{\tau}' \rangle, \ldots \textbf{ satisfies } \bullet \ \{\bullet\} \in \Psi \qquad \Psi \mid \Theta \vdash_I \vec{\tau}_1, I\langle \vec{\tau}' \lfloor^{\vec{\tau}}/_{\Theta'} \rceil \rangle, \vec{\tau}_2 <: \langle \vec{\tau}^a \rangle \rightsquigarrow \langle \Theta^{<:} \rangle}{\Psi \mid \Theta \vdash_I \vec{\tau}_1, I'\langle \vec{\tau} \rangle, \vec{\tau}_2 <: \langle \vec{\tau}_I^a \rangle \rightsquigarrow \langle \Theta^{<:} \rangle}$$

$$\frac{}{\Psi \mid \Theta \vdash_I \vec{\tau}_1, \bot, \vec{\tau}_2 <: \langle \vec{\tau}_I^a \rangle \rightsquigarrow \langle \rangle} \qquad \frac{\Psi \mid \Theta \vdash_I \vec{\tau}_1, \vec{\tau}_2 <: \langle \vec{\tau}_I^a \rangle \rightsquigarrow \langle \Theta^{<:} \rangle}{\Psi \mid \Theta \vdash_I \vec{\tau}_1, \top, \vec{\tau}_2 <: \langle \vec{\tau}_I^a \rangle \rightsquigarrow \langle \Theta^{<:} \rangle}$$

$$\frac{\Psi \mid \Theta \vdash_I \vec{\tau}_1, \tau_1, \vec{\tau}_2 <: \langle \vec{\tau}_I^a \rangle \rightsquigarrow \langle \Theta_1^{<:} \rangle \qquad \Psi \mid \Theta \vdash_I \vec{\tau}_1, \tau_2, \vec{\tau}_2 <: \langle \vec{\tau}_I^a \rangle \rightsquigarrow \langle \Theta_2^{<:} \rangle}{\Psi \mid \Theta \vdash_I \vec{\tau}_1, \tau_1 \cup \tau_2, \vec{\tau}_2 <: \langle \vec{\tau}_I^a \rangle \rightsquigarrow \langle \Theta_1^{<:}, \Theta_2^{<:} \rangle}$$

$$\frac{\Psi \mid \Theta \vdash_I \vec{\tau}_1, \tau_1, \tau_2, \vec{\tau}_2 <: \langle \vec{\tau}_I^a \rangle \rightsquigarrow \langle \Theta^{<:} \rangle}{\Psi \mid \Theta \vdash_I \vec{\tau}_1, \tau_1 \cap \tau_2, \vec{\tau}_2 <: \langle \vec{\tau}_I^a \rangle \rightsquigarrow \langle \Theta^{<:} \rangle}$$

Intersected Type-Arguments Bound Inference $\Psi \mid \Theta \vdash_{\langle \Theta \rangle} \langle \vec{\tau} \rangle, \ldots <: \langle \vec{\tau}^a \rangle \rightsquigarrow \langle \Theta^{<:} \rangle$

$$\frac{}{\Psi \mid \Theta \vdash_{\langle \rangle} \langle \rangle, \ldots <: \langle \rangle \rightsquigarrow \langle \rangle}$$

$$\frac{\Psi \mid \Theta \vdash_{\langle \Theta' \rangle} \langle \vec{\tau}_1 \rangle, \ldots <: \langle \vec{\tau}^a \rangle \rightsquigarrow \langle \Theta^{<:} \rangle \qquad \Psi \mid \Theta \vdash_\nu \tau_1, \ldots <: \tau^a \rightsquigarrow \langle \Theta_\alpha^{<:} \rangle}{\Psi \mid \Theta \vdash_{\langle \Theta', \bullet <:\nu\alpha<: \bullet \rangle} \langle \vec{\tau}_1, \tau_1 \rangle, \ldots <: \langle \vec{\tau}^a, \tau^a \rangle :\rightsquigarrow \langle \Theta^{<:}, \Theta_\alpha^{<:} \rangle}$$

Intersected Type-Argument Bound Inference $\Psi \mid \Theta \vdash_\nu \tau, \ldots <: \tau^a \rightsquigarrow \langle \Theta^{<:} \rangle$

$$\frac{\Psi \mid \Theta \vdash \tau_1 \cap \ldots <: \tau \rightsquigarrow \langle \Theta^{<:} \rangle}{\Psi \mid \Theta \vdash_+ \tau_1, \ldots <: \tau \rightsquigarrow \langle \Theta^{<:} \rangle} \qquad \frac{\Psi \mid \Theta \vdash \tau <: \tau_1 \cup \ldots \rightsquigarrow \langle \Theta^{<:} \rangle}{\Psi \mid \Theta \vdash_- \tau_1, \ldots <: \tau \rightsquigarrow \langle \Theta^{<:} \rangle}$$

$$\frac{\Psi \mid \Theta \vdash \tau <: \tau_1 \cup \ldots \rightsquigarrow \langle \Theta_-^{<:} \rangle \qquad \Psi \mid \Theta \vdash \tau_1 \cap \ldots <: \tau \rightsquigarrow \langle \Theta_+^{<:} \rangle}{\Psi \mid \Theta \vdash_! \tau_1, \ldots <: \tau \rightsquigarrow \langle \Theta_-^{<:}, \Theta_+^{<:} \rangle}$$

$$\frac{\Psi \mid \Theta \vdash \tau_i <: \tau_1 \cup \ldots \rightsquigarrow \langle \Theta_i^{<:} \rangle \qquad \Psi \mid \Theta \vdash \tau_1 \cap \ldots <: \tau_o \rightsquigarrow \langle \Theta_o^{<:} \rangle}{\Psi \mid \Theta \vdash_! \tau_1, \ldots <: \textbf{in } \tau_i \textbf{ out } \tau_o \rightsquigarrow \langle \Theta_i^{<:}, \Theta_o^{<:} \rangle}$$

Figure 8.7 (contd.)

As an example of an unsatisfiable premise, suppose the interface `Values` extends $\texttt{Seq}\langle\top\rangle$. The type $\top$ is not equatable with anything; if it were it would defeat the whole point of trying for type-safe equality. Consequently, `Values` should not need to be equatable since its elements are not. This is reflected in the fact that $\Psi \mid \varnothing \mid \varsigma : \top.\texttt{Eq}\langle\top\rangle \mid \varnothing \vdash \bot$ is provable in our system, as formalized in Figure 8.7. In particular, since $\top$ satisfies no shapes, the evidence ς is necessarily unconstructable and so the assumed premise is in fact unsatisfiable.

Even if the premise is satisfiable, our system is able to infer some useful conclusions from it. For example, suppose $\texttt{Graph}\langle\texttt{L}\rangle$ extends $\texttt{Set}\langle\texttt{Node}\langle\texttt{L}\rangle\rangle$, where $\texttt{Set}\langle\texttt{E}\rangle$ is equatable with itself (using set equality) if `E` is, and similarly $\texttt{Node}\langle\texttt{L}\rangle$ is equatable with itself if `L` is. Because $\texttt{Graph}\langle\texttt{L}\rangle$ extends $\texttt{Set}\langle\texttt{Node}\langle\texttt{L}\rangle\rangle$, it must be equatable with other sets of nodes provided $\texttt{Node}\langle\texttt{L}\rangle$ is equatable with itself. Using the judgement $\Psi \mid \Theta \mid \Sigma^\tau \mid \varnothing \vdash \langle\Theta'\rangle[\Sigma']$, our system is able to infer from the fact that $\texttt{Node}\langle\texttt{L}\rangle$ is presumed to be equatable with itself (i.e. $\varsigma : \texttt{Node}\langle\texttt{L}\rangle.\texttt{Eq}\langle\texttt{Node}\langle\texttt{L}\rangle\rangle$ is in Σ^τ), then the only way this can be true given the definition of `Node` is if `L` is in fact equatable with itself (i.e. and so $\varsigma' : \texttt{L.Node}\langle\texttt{L}\rangle$ can be in Σ'). Thus $\texttt{Graph}\langle\texttt{L}\rangle$ can satisfy its conditional shape inherited from $\texttt{Seq}\langle\texttt{Node}\langle\texttt{L}\rangle\rangle$ via the declaration **`satisfies`** $\texttt{Eq}\langle\texttt{Seq}\langle\texttt{Node}\langle\texttt{L}\rangle\rangle\rangle(\varsigma : \texttt{L.Eq}\langle\texttt{L}\rangle)$.

Note that this reasoning relies on the fact that the only way evidence can be constructed is through the declarations specified by interfaces. This is not the case in systems that use implicit arguments [Odersky, Altherr, et al., 2014, Chapter 7] or models [Zhang, Loring, et al., 2015] to address the binary-method problem. The advantage of those systems is that programmers can supply alternative notions of equality, such as case-insensitive equality on strings, and expect reasonable support for

this common customization. The disadvantage is that implementations of, say, a `contains` method cannot reliably optimize for the standard notion of equality because they cannot be sure which notion of equality the implementation of `contains` will need to support. For example, an implementation of `Graph⟨Integer⟩` might want to take advantage of the fact that its node-set is backed by a hashmap from integer labels to their respective nodes. In our system it can, but in systems that permit alternative evidence this implementation must always fall back on inefficient general-purpose algorithms in case the given notion of equality on integers is, say, modulo some prime number. Thus there is a design tradeoff here, and it would be interesting future work to design a language that can conveniently combine the functionalities and insights of both alternatives.

8.5 METHOD SIGNATURES AND TYPE-ARGUMENT INFERABILITY

With the infrastructure for types, subtyping, shapes, and shape satisfaction in place, we now proceed to the final pass of hierarchy validation: method signatures. This pass ensures that all method signatures have the property that, for any given arguments, type arguments can always be inferred to produce a principal return type (if there is a valid set of type arguments at all). Using that property, this pass also ensures that interfaces provide the methods required by the declared inherited interfaces and conditionally satisfied shapes. (For simplicity, the formalization requires interfaces duplicate rather than simply reuse inherited method signatures.) Again, the design of method signatures is nearly identical for the two calculi.

Method Name m Program Variable x Function Variable f

$$
\begin{array}{lll}
\text{Method Signature} & s & ::= m\langle\Theta^!\rangle[\Theta](\Gamma)[\Sigma] : \tau \\
\text{Program Context} & \Gamma & ::= p; ... \\
\text{Program Parameter} & p & ::= x : \tau \mid f(\tau, ...) : \tau \\
\text{Inhabitation} & \iota & ::= \uparrow \mid \downarrow
\end{array}
$$

Signatures Validity $\Psi \vdash^s \Psi$

$$
\frac{}{\Psi \vdash^s \varnothing} \qquad \frac{\Psi \vdash^s \Psi' \qquad \Psi \mid \Theta \mid \Sigma \vdash s_1 \qquad ... \qquad \Psi \mid \Theta \mid \varnothing \vdash \{s_1; ...\} \sqsubseteq \sigma_1 \qquad ...}{\Psi \vdash^s \Psi'; \texttt{shape}\ S\langle\Theta\rangle\ \texttt{extends}\ \sigma_1, ...\ \{s_1; ...\}}
$$

$$
\frac{\Psi \vdash^s \Psi' \qquad \Psi \mid \Theta \mid \varnothing \vdash s_1 \qquad ... \qquad \Psi \mid \Theta \mid \varnothing \vdash \{s_1; ...\} \sqsubseteq \vec{\tau}^I \qquad \Psi \mid \Theta, \Theta_1 \mid \Sigma_1 \vdash \{s_1; ...\} \sqsubseteq \sigma_1 \qquad ...}{\Psi \vdash^s \Psi'; \texttt{interface}\ I\langle\Theta\rangle\ \texttt{extends}\ \vec{\tau}^I\ \texttt{satisfies}\ \sigma_1[\Theta_1][\Sigma_1], ...\ \{s_1; ...\}}
$$

Method-Signature Validity $\Psi \mid \Theta \mid \Sigma \vdash s$

$$
\frac{\begin{array}{c}\Psi \mid \Theta \vdash \langle \Theta_e^!, \Theta_i \rangle \qquad \vdash \langle \Theta_i \rangle \rightsquigarrow \langle \Theta_i^! \rangle \\ \Psi \mid \Theta, \Theta_e^!, \Theta_i^! \vdash (\Gamma_m) \qquad \Psi \mid \Theta, \Theta_e^!, \Theta_i^! \mid \Sigma \vdash [\Sigma_m] \qquad \Psi \mid \Theta, \Theta_e^!, \Theta_i^! \vdash_+ \tau_m \\ \vdash \langle \Theta \rangle \rightsquigarrow \langle \Theta^! \rangle \qquad \Psi \mid \Theta^!, \Theta_e^! \vdash (\Gamma_m)[\Sigma_m] \rightsquigarrow \langle \Theta_i \mid \varnothing \rangle\end{array}}{\Psi \mid \Theta \mid \Sigma \vdash m\langle\Theta_e^!\rangle[\Theta_i](\Gamma_m)[\Sigma_m] : \tau_m}
$$

Method-Signature Inferability $\Psi \mid \Theta \vdash (\Gamma)[\Sigma] \rightsquigarrow \langle \Theta \mid \Theta^! \rangle$

$$
\frac{}{\Psi \mid \Theta^! \vdash ()[] \rightsquigarrow \langle \varnothing \mid \Theta_u^! \rangle}
$$

$$
\frac{\Psi \mid \Theta^! \vdash (\Gamma)[\Sigma] \rightsquigarrow \langle \Theta_i \mid \Theta_u^! \rangle \qquad \Psi \mid \Theta^!, \Theta_i, \Theta_u^! \vdash_- \tau}{\Psi \mid \Theta^! \vdash (\Gamma; x : \tau)[\Sigma] \rightsquigarrow \langle \Theta_i \mid \Theta_u^! \rangle}
$$

$$
\frac{\begin{array}{c}\Psi \mid \Theta^! \vdash (\Gamma)[\Sigma] \rightsquigarrow \langle \Theta_i \mid \Theta_u^! \rangle \\ \Psi \mid \Theta^!, \Theta_i \vdash_+ \tau_1 \qquad ... \qquad \Psi \mid \Theta^!, \Theta_i, \Theta_u^! \vdash_- \tau\end{array}}{\Psi \mid \Theta^! \vdash (\Gamma; f(\tau_1, ...) : \tau)[\Sigma] \rightsquigarrow \langle \Theta_i \mid \Theta_u^! \rangle}
$$

$$
\frac{\Psi \mid \Theta^! \vdash (\Gamma)[\Sigma] \rightsquigarrow \langle \Theta_i \mid \Theta_u^! \rangle \qquad \Theta^!, \Theta_i \vdash_+ \alpha \qquad \Psi \mid \Theta^!, \Theta_i, \Theta_u^! \vdash_- \sigma}{\Psi \mid \Theta^! \vdash (\Gamma)[\Sigma, \varsigma : \alpha.\sigma] \rightsquigarrow \langle \Theta_i \mid \Theta_u^! \rangle}
$$

$$
\frac{\begin{array}{c}\Psi \mid \Theta^! \vdash (\Gamma)[\Sigma] \rightsquigarrow \langle \Theta_i \mid \tau_\ell <: !\alpha <: \tau_u, \Theta_u^! \rangle \\ \Psi \mid \Theta^!, \Theta_i \vdash_+ \tau_\ell \qquad \Psi \mid \Theta^!, \Theta_i \vdash_- \tau_u \\ \vdash \langle \Theta_i \rangle \rightsquigarrow \langle \Theta_i^! \rangle \qquad \Psi \mid \Theta^!, \Theta_i^!, \tau_\ell <: \nu\alpha <: \tau_u, \Theta_u^! \vdash (\Gamma)[\Sigma] \rightsquigarrow \nu\alpha\end{array}}{\Psi \mid \Theta^! \vdash (\Gamma)[\Sigma] \rightsquigarrow \langle \Theta_i, \tau_\ell <: \nu\alpha <: \tau_u \mid \Theta_u^! \rangle}
$$

Figure 8.8: Method Signatures

Principal Inferability $\Psi \mid \Theta \vdash (\Gamma)[\Sigma] \rightsquigarrow \nu\alpha$

$$\frac{x : \tau \in \Gamma \qquad \Psi \mid \Theta \vdash +\tau\downarrow \rightsquigarrow !\alpha\uparrow}{\Psi \mid \Theta \vdash (\Gamma)[\Sigma] \rightsquigarrow !\alpha}$$

$$\frac{x : \tau \in \Gamma \qquad \varsigma : \alpha'.\sigma \in \Sigma \qquad \Psi \mid \Theta \vdash +\tau\downarrow \rightsquigarrow \nu\alpha'\downarrow \qquad \Psi \mid \Theta \vdash +\sigma\downarrow \rightsquigarrow !\alpha\uparrow}{\Psi \mid \Theta \vdash (\Gamma)[\Sigma] \rightsquigarrow !\alpha}$$

$$\frac{}{\Psi \mid \Theta \vdash (\Gamma)[\Sigma] \rightsquigarrow \nu^{\pm}\alpha}$$

$$\frac{\Psi \mid \Theta \vdash (p_1) \rightsquigarrow \nu^{\pm}\alpha \quad \ldots \qquad \Psi \mid \Theta \vdash +\sigma_1\uparrow \rightsquigarrow \nu^{\pm}\alpha\uparrow \quad \ldots}{\Psi \mid \Theta \vdash (p_1, \ldots)[\alpha_1.\sigma_1, \ldots] \rightsquigarrow \nu^{\pm}\alpha}$$

Parameter Inferability $\Psi \mid \Theta \vdash (p) \rightsquigarrow \nu^{\pm}\alpha$

$$\frac{\Psi \mid \Theta \vdash +\tau\downarrow \rightsquigarrow \nu^{\pm}\alpha\uparrow}{\Psi \mid \Theta \vdash (x : \tau) \rightsquigarrow \nu^{\pm}\alpha} \qquad \frac{\Psi \mid \Theta \vdash +\tau\uparrow \rightsquigarrow \nu^{\pm}\alpha\uparrow}{\Psi \mid \Theta \vdash (f(\tau_1, \ldots) : \tau) \rightsquigarrow \nu^{\pm}\alpha}$$

Type/Shape Inferability $\Psi \mid \Theta \vdash \nu\tau^m\iota \rightsquigarrow \nu\alpha\iota$

$$\frac{\textbf{interface } I\langle\Theta_I\rangle \textbf{ extends } \bullet \textbf{ satisfies } \bullet \in \Psi \qquad \vdash \nu \leq + \qquad \Psi \mid \Theta \vdash \nu\langle\vec{\tau}^a\rangle\iota : \langle\Theta_I\rangle \rightsquigarrow \nu_\alpha\alpha}{\Psi \mid \Theta \vdash \nu I\langle\vec{\tau}^a\rangle\iota \rightsquigarrow \nu_\alpha\alpha\uparrow} \qquad \frac{\Psi \mid \Theta \vdash_{-*\nu^{\pm}*\nu} \tau}{\Psi \mid \Theta \vdash \nu\tau\iota \rightsquigarrow \nu^{\pm}\alpha\uparrow}$$

$$\frac{\textbf{shape } S\langle\Theta_S\rangle \textbf{ extends } \bullet \in \Psi \qquad \Psi \mid \Theta \vdash +\langle\vec{\tau}^a\rangle\iota : \langle\Theta_S\rangle \rightsquigarrow \nu_\alpha\alpha}{\Psi \mid \Theta \vdash +S\langle\vec{\tau}^a\rangle\iota \rightsquigarrow \nu_\alpha\alpha\uparrow} \qquad \frac{\vdash \nu \leq \nu_\alpha \qquad \iota \vdash \iota_\alpha}{\Psi \mid \Theta \vdash \nu\alpha\iota \rightsquigarrow \nu_\alpha\alpha\iota_\alpha}$$

$$\frac{\Psi \mid \Theta \vdash +\tau_1\iota \rightsquigarrow \nu_\alpha\alpha\iota_1 \qquad \Psi \mid \Theta \vdash +\tau_2\iota \rightsquigarrow \nu_\alpha\alpha\iota_2 \qquad \iota_1, \iota_2 \vdash \iota_\alpha}{\Psi \mid \Theta \vdash +(\tau_1 \cap \tau_2)\iota \rightsquigarrow \nu_\alpha\alpha\iota_\alpha} \qquad \frac{\Psi \mid \Theta \vdash -\tau_1\uparrow \rightsquigarrow \nu^{\pm}\alpha\uparrow \qquad \Psi \mid \Theta \vdash -\tau_2\uparrow \rightsquigarrow \nu^{\pm}\alpha\uparrow}{\Psi \mid \Theta \vdash -(\tau_1 \cup \tau_2)\uparrow \rightsquigarrow \nu^{\pm}\alpha\uparrow}$$

Type-Arguments Inferability $\Psi \mid \Theta \vdash \nu\langle\vec{\tau}^a\rangle\iota : \langle\Theta\rangle \rightsquigarrow \nu\alpha$

$$\frac{}{\Psi \mid \Theta \vdash \nu\langle\rangle\iota : \langle\rangle \rightsquigarrow \nu^{\pm}\alpha}$$

$$\frac{\Psi \mid \Theta \vdash \nu\langle\vec{\tau}^a\rangle\iota : \langle\Theta'\rangle \rightsquigarrow \nu^{\pm}\alpha \qquad \Psi \mid \Theta \vdash (\nu' * \nu)\tau\uparrow \rightsquigarrow \nu^{\pm}\alpha\uparrow}{\Psi \mid \Theta \vdash \nu\langle\vec{\tau}^a, \tau\rangle\iota : \langle\Theta', \bullet <: \nu'\alpha' <: \bullet\rangle \rightsquigarrow \nu^{\pm}\alpha}$$

$$\frac{\Psi \mid \Theta \vdash \nu\langle\vec{\tau}^a\rangle\iota : \langle\Theta'\rangle \rightsquigarrow \nu^{\pm}\alpha \qquad \Psi \mid \Theta \vdash (- * \nu)\tau_i\uparrow \rightsquigarrow \nu^{\pm}\alpha\uparrow \qquad \Psi \mid \Theta \vdash \nu\tau_o\uparrow \rightsquigarrow \nu^{\pm}\alpha\uparrow}{\Psi \mid \Theta \vdash \nu\langle\vec{\tau}^a, \textbf{in } \tau_i \textbf{ out } \tau_o\rangle\iota : \langle\Theta', \bullet <: !\alpha' <: \bullet\rangle \rightsquigarrow \nu^{\pm}\alpha}$$

$$\frac{i \in \{1, \ldots\} \qquad \Psi \mid \Theta \vdash (\nu_i * \nu)\tau_i^a\uparrow \rightsquigarrow !\alpha\uparrow}{\Psi \mid \Theta \vdash \nu\langle\tau_1^a, \ldots\rangle\downarrow : \langle\bullet <: \nu_1\alpha_1 <: \bullet, \ldots\rangle \rightsquigarrow !\alpha}$$

Figure 8.8 (contd.)

Inhabitation $\iota, \ldots \vdash \iota$

$$\frac{}{\iota, \ldots \vdash \uparrow} \qquad \frac{}{\ldots, \downarrow, \ldots \vdash \downarrow}$$

Interface Inheritance

$$\frac{\Psi \mid \Theta \mid \Sigma \vdash \{s; \ldots\} \sqsubseteq \vec{\tau}^I}{\Psi \mid \Theta \mid \Sigma \vdash \{s; \ldots\} \sqsubseteq \tau^I} \qquad \frac{}{\Psi \mid \Theta \mid \Sigma \vdash \{s_1; \ldots\} \sqsubseteq \varnothing} \qquad \frac{\Psi \mid \Theta \mid \Sigma \vdash \{s_1; \ldots\} \sqsubseteq \tau_1^I \quad \ldots}{\Psi \mid \Theta \mid \Sigma \vdash \{s_1; \ldots\} \sqsubseteq \tau_1^I, \ldots}$$

$$\frac{\textbf{interface}\ I\langle\Theta_I\rangle\ \textbf{extends}\ \bullet\ \textbf{satisfies}\ \bullet \quad \{s_1'; \ldots\} \in \Psi \qquad \Psi \mid \Theta \mid \Sigma \vdash \{s_1; \ldots\} \sqsubseteq s_1'\lfloor^{\vec{\tau}}/_{\Theta_I}\rceil \quad \ldots}{\Psi \mid \Theta \mid \Sigma \vdash \{s_1; \ldots\} \sqsubseteq I\langle\vec{\tau}\rangle}$$

Shape Inheritance $\Psi \mid \Theta \mid \Sigma \vdash \{s; \ldots\} \sqsubseteq \sigma$

$$\frac{\textbf{shape}\ S\langle\Theta_S\rangle\ \textbf{extends}\ \bullet \quad \{s_1'; \ldots\} \in \Psi \qquad \Psi \mid \Theta \mid \Sigma \vdash \{s_1; \ldots\} \sqsubseteq s_1'\lfloor^{\vec{\tau}}/_{\Theta_S}\rceil \quad \ldots}{\Psi \mid \Theta \mid \Sigma \vdash \{s_1; \ldots\} \sqsubseteq S\langle\vec{\tau}\rangle}$$

Method Inheritance $\Psi \mid \Theta \mid \Sigma \vdash \{s; \ldots\} \sqsubseteq s^\tau$

$$\frac{\Psi \mid \Theta, \Theta_e^!, \Theta_i \mid \Sigma, \Sigma^\tau \vdash \bot}{\Psi \mid \Theta \mid \Sigma \vdash \{s_1; \ldots\} \sqsubseteq m\langle\Theta_e^!\rangle[\Theta_i](\Gamma)[\Sigma^\tau] : \tau}$$

$$\frac{\begin{array}{c}\Psi \mid \Theta, \Theta_2^!, \Theta_2 \mid \Sigma, \Sigma_2^\tau \vdash \langle\Theta'\rangle[\Sigma'] \\ \Theta_2^! = \bullet <: \nu_1\alpha_1 <: \bullet, \ldots \qquad \Psi \mid \Theta' \vdash_? \langle\alpha_1, \ldots\rangle : \langle\Theta_1^!\rangle \\ \Psi \mid \Theta' \vdash_? \langle\vec{\tau}\rangle : \langle\Theta_1\lfloor^{\alpha_1,\ldots}/_{\Theta_1^!}\rceil\rangle \qquad \Psi \mid \Theta' \vdash (\Gamma_2) <: (\Gamma_1\lfloor^{\alpha_1,\ldots,\vec{\tau}}/_{\Theta_1^!,\Theta_1}\rceil) \\ \Psi \mid \Theta' \mid \Sigma' \vdash \Sigma_1\lfloor^{\alpha_1,\ldots,\vec{\tau}}/_{\Theta_1^!,\Theta_1}\rceil \qquad \Psi \mid \Theta' \vdash \tau_1\lfloor^{\alpha_1,\ldots,\vec{\tau}}/_{\Theta_1^!,\Theta_1}\rceil <: \tau_2\end{array}}{\Psi \mid \Theta \mid \Sigma \vdash \{\ldots; m\langle\Theta_1^!\rangle[\Theta_1](\Gamma_1)[\Sigma_1] : \tau_1; \ldots\} \sqsubseteq m\langle\Theta_2^!\rangle[\Theta_2](\Gamma_2)[\Sigma_2^\tau] : \tau_2}$$

Program Context Validity $\Psi \mid \Theta \vdash (\Gamma) \qquad \Psi \mid \Theta \vdash (p)$

$$\frac{\Psi \mid \Theta \vdash (p_1) \quad \ldots}{\Psi \mid \Theta \vdash (p_1, \ldots)} \qquad \frac{\Psi \mid \Theta \vdash_- \tau}{\Psi \mid \Theta \vdash (x : \tau)} \qquad \frac{\Psi \mid \Theta \vdash_+ \tau_1 \quad \ldots \quad \Psi \mid \Theta \vdash_- \tau}{\Psi \mid \Theta \vdash (f(\tau_1, \ldots) : \tau)}$$

Program-Context Subtyping $\Psi \mid \Theta \vdash (\Gamma) <: (\Gamma) \qquad \Psi \mid \Theta \vdash (p) <: (p)$

$$\frac{\Psi \mid \Theta \vdash (p_1) <: (p_1') \quad \ldots}{\Psi \mid \Theta \vdash (p_1, \ldots) <: (p_1', \ldots)} \qquad \frac{\Psi \mid \Theta \vdash \tau <: \tau'}{\Psi \mid \Theta \vdash (x : \tau) <: (x : \tau')}$$

$$\frac{\Psi \mid \Theta \vdash \tau_1' <: \tau_1 \quad \ldots \quad \Psi \mid \Theta \vdash \tau <: \tau'}{\Psi \mid \Theta \vdash (f(\tau_1, \ldots) : \tau) <: (f(\tau_1', \ldots) : \tau')}$$

Figure 8.8 (contd.)

8.5.1 *Method Signatures*

As shown in Figure 8.8, a method signature s is comprised of 6 parts: the method name m, the explicit type parameters $\Theta^!$, the inferable type parameters Θ, the explicit program parameters Γ, the inferable shape evidence Σ, and the return type τ. In practice, many of these components will often be empty and simply omitted. When invoking a method, the inferable components will *always* be omitted because they will always be guaranteed to be unambiguously inferable. So while the formalism is complex, in practice much of this complexity is optional and will be forgone except for precisely the cases that truly need it.

Notice that the grammar indicates that each inferable type parameter has a specified variance. In practice, these variances can be inferred in all but a few cases from how these type parameters are used elsewhere in the signature, but for simplicity we make them explicit in our formalism. The variance of an inferable type parameter specifies which inference strategy to employ. For invariant type parameters, the signature must ensure that the corresponding type argument will always be *uniquely* determined by the types of the arguments. For covariant type parameters, constraints will be collected and then the inferred type argument will be the *minimal* type satisfying those constraints (if any). For contravariant type parameters, upper-bound constraints will be collected and then the inferred type argument will be the *maximal* type satisfying those constraints (if any).

In validating a method signature, the majority of the work is checking that the various components are valid and have the appropriate variance with respect to the type variables in the outer context, as is standard.

Notice that the return type must be covariant with respect to the declared variances on the inferable type parameter in addition to the outer context. This means that the above inference strategies are guaranteed to produce the most precise return type possible, thus ensuring a principal type for any invocation of this signature.

With these strategies in mind, method-signature validity also checks that the type parameters are principally inferable from the inputs to the method. This check is done using the judgement $\Psi \mid \Theta \vdash (\Gamma)[\Sigma] \rightsquigarrow \langle \Theta \mid \Theta^{!} \rangle$. The purpose of this judgement is to ensure that there will be a principal type argument for each type parameter given their declared inference strategies, and to ensure that there is a computable sequential process for determining these principal type arguments. But before we discuss inferability, we must first discuss another oddity of our calculi.

8.5.2 *Higher-Order Parameters*

Lambda expressions have become omnipresent in major nominal object-oriented languages. As such, any practical calculus for generics needs to support them, or at least support the concept. Unfortunately, they pose a key challenge for decidable type-checking. In particular, practicality seems to necessitate handling lambda expressions with unannotated parameter types. And while languages without subtyping can overcome this challenge by using unification [Hindley, 1969; Milner, 1978], the problem is in general undecidable for languages with subtyping [Pierce, 1992].

We address this by making lambdas be *arguments* but not *expressions*. That is, a method can specify a higher-order parameter $f(\tau_1, ...) : \tau$, and

an invocation can then provide a (named) argument $f(x, ...) \mapsto e$. This compromise seems to be practical enough, given that C# and Java both effectively restrict the use of lambda expressions to when such context is available [ECMA TC39-TG2, 2017, §12.16] [Gosling, Joy, Steele, Bracha, and Buckley, 2015, §15.27]. In fact, both Kotlin and Ceylon have adopted a specialized syntax for higher-order parameters very much like our own [JetBrains, 2019, Higher-Order Functions and Lambdas] [King, 2013, §1.3.4]. This syntax has the added advantage that, since higher-order parameters are not themselves values (e.g. do not have type $\top$), the closures providing higher-order arguments can be allocated on the stack by default rather than the heap, making higher-order method invocation much more efficient.

This syntax also makes it clear that higher-order parameters interact differently with inference than do functional interfaces. For example, with the higher-order parameter $f(\alpha) : \beta$, the type variable α needs to be covariantly (or invariantly) inferred so that a minimal type argument for α provides a most-precise input type to the argument corresponding to f, which can then be used to infer the type of β. On the other hand, a parameter of type `Fun`$\langle\alpha, \beta\rangle$ can even be used to constrain a contravariantly inferred type parameter α because the corresponding argument's type will necessarily have a known principal instantiation of `Fun`.

Another important constraint on using higher-order parameters is that the input types must be completely inferable without the output type being known. For example, the signature `repeat`$\langle$`T`$\rangle$`(init: T; step(T): T): T` in interface `Nat` is uninferable (regardless of what inference strategy one uses for `T`). To see why, suppose `nat` is a `Nat` and consider the invocation `nat.repeat(init` $\mapsto$ `empty; step(s)` $\mapsto$ `singleton(s))`, where

`empty` has type $\texttt{Seq}\langle\bot\rangle$, and where `singleton` has signature $[+\alpha](\alpha) : \texttt{Seq}\langle\alpha\rangle$. One could infer `T` to be $\texttt{Seq}\langle\top\rangle$, but one could also infer `T` to be $\texttt{Seq}\langle\texttt{Seq}\langle\top\rangle\rangle$, which results in a more-precise return type. In fact, this generalizes to an infinite chain of more-precise type arguments for `T`, but there is no valid type argument that lower-bounds this chain. As such, it is not simply that we cannot figure out how to infer the type argument for `repeat`; there in fact is *no* principal type argument for `repeat`. So in our system, `T` would have to be changed to an explicit type parameter. Nonetheless, many other common and important examples like `map` have completely inferable type parameters.

8.5.3 *Inferability*

Now we discuss the judgement $\Psi \mid \Theta \vdash (\Gamma)[\Sigma] \rightsquigarrow \langle\Theta \mid \Theta^!\rangle$, which ensures that one can always infer principal type arguments for the inferable type parameters from the given program arguments due to the structure of the program parameters Γ and shape evidence Σ. The first rule of this judgement indicates that the validation process begins with no parameters, no shape constraints, and all the type parameters in the uninferred bin to the right. The remaining rules then process one component of the signature at a time, until eventually the entire signature is determined to be inferable.

The simplest component to add is a first-order parameter $x : \tau$. The one requirement is that the corresponding argument for this parameter should not impose any new constraints that might have affected the type parameters that have already been inferred. This is ensured by checking

that τ is contravariant with respect to those type parameters. Note that the corresponding argument can still affect whether or not the inferred type arguments are *valid*, but the contravariance of τ ensures that if the type arguments are now made to be invalid, then there necessarily are no type arguments that could be valid, and so the method invocation must be rejected.

When adding a higher-order parameter $f(\tau_1, ...) : \tau$, we first ensure that the input types τ_1 are covariant with respect to the inferred type parameters and make no reference to the uninferred type parameters. This means that, given the corresponding higher-order argument, we can infer its return type from these inferred input types and use that to constrain the remaining uninferred type parameters. So lastly we ensure that the return type τ does not introduce any new constraints on the already inferred type parameters.

When adding shape evidence $\varsigma : \alpha.\sigma$, we first ensure that α has already been inferred. This allows us to determine its corresponding principal instantiation of σ, whose determination we then ensure does not impose new constraints on the already inferred type parameters.

When inferring a type parameter α, we first ensure that its bounds are easier to satisfy the more precisely we infer type arguments for the already inferred type parameters. The judgement $\Psi \mid \Theta \vdash (\Gamma)[\Sigma] \rightsquigarrow \nu\alpha$ then ensures that α is principally inferable based on how it occurs in the determined argument and evidence types. For invariant type parameters, α must occur in a uniquely-determining position of a type that is guaranteed to be inhabited when the method is called. (Unfortunately, α cannot be inferred from possibly-uninhabited types due to the possibility of $\bot$ being nested in the corresponding type argument). For co/contravariant

arguments, we must check that the given argument and evidence types unambiguously constrain α on the appropriate side. For Gen_{α}, this is always the case, but for $\mathrm{Gen}_{\cap}$ the presence of arbitrary union and intersection types can cause ambiguities.

Altogether, the proof search for this judgement is decidable, and the resulting proof precisely informs the type-checker and run-time system how to determine the necessary type arguments, infer the parameters and return types of the higher-order arguments, and construct the required shape evidence. Note that multiple proofs are possible, resulting in different inference strategies, but all these strategies will result in the same conclusion, thereby providing semantic coherence as needed for gradual typing and for intersecting return types.

8.5.4 *Inheritance*

After checking that all the method signatures of a given interface or shape are inferable, we then use judgements $\Psi \mid \Theta \mid \Sigma \vdash \{s_1; ...\} \sqsubseteq \vec{\tau}^I/\sigma$ to check that these signatures subsume the methods of inherited interfaces or (conditionally satisfied) shapes. For this, we simply check whether the invocation of the method signature would succeed given the environment provided by the inherited method signature, relying on the fact that type-argument inference for the method signature has already been guaranteed to be decidable. This enables our system to be much more flexible in how subinterfaces satisfy the requirements of superinterfaces.

For example, consider the following hierarchy (using shorthands):

```
shape Eq⟨−T⟩ { equals(T): Bool;}
interface Num satisfies Eq⟨Num⟩{ equals(Num): Bool;...;}
interface Int extends Num satisfies Eq⟨Num⟩ { equals(Num): Bool;...;}
interface Set⟨+E⟩ { contains[+F:>E](F)[F.Eq⟨F⟩]: Bool;...;}
interface BitSet extends Seq⟨Int⟩ { contains(Num): Bool;... }
```

Besides `BitSet`, this hierarchy is fairly standard among related work on generics. The interface `Set` might seem to have a complex signature for `contains`, but this complexity is necessary for two reasons. First, we want `Set` to be covariant, which means it cannot have the method `contains(E)`. Second, one cannot properly implement `contains` without a proper notion of equality. Parameterizing the method by a type variable lower-bounded by `E` solves the first problem, and adding the `Eq` constraint or implicit/inferred argument solves the second problem.

What is surprising is that `BitSet`'s signature for `contains` is allowed to forgo all this complexity despite inheriting `Set`. This is for two reasons.

The first is that the inferred type arguments and constructed shape evidence are not actually run-time arguments in our system; type arguments are merely guaranteed to exist and can be constructed by the *callee* (rather than the caller) *if needed*, and shape evidence merely guarantees that methods can be invoked on instances of the appropriate type. This *dynamic* inference is in fact necessary to address the problem we identified with `Singleton` in Section 8.2.4 so that we can ensure semantic coherence for gradual typing. And since many type parameters are inferable, this means that our system can reify types much less frequently than prior systems with reifable generics.

The second is that, because the type parameter `F` of the inherited method is a supertype of `E`, which is `Int` for `BitSet`, and because `F` is assumed to satisfy `Eq⟨F⟩`, our shape-simplification judgement can in fact infer that

`F` must be also be a *sub*type of `Num`. This is because `F`'s lower bound, `Int`, principally satisfies `Eq⟨Num⟩`, guaranteeing that `Eq⟨Num⟩` is a subshape of `Eq⟨F⟩`, which by contravariance can only be true if `F` is a subtype of `Num`. Consequently, the inherited signature's parameter of type `F` is necessarily a valid argument of type `Num`.

So by combining insights from decidable type-argument inference and the (dynamic) gradual guarantee, our system is able to permit `BitSet` to have a simple, intuitive, and type-safe signature for `contains` that furthermore supports the obvious efficient implementation a bit-set can provide for containment of numbers with the standard notion of equality.

8.5.5 *Practicality*

One might wonder whether the inferability requirements we impose are practical. We have evaluated their practicality in two ways. For our first evaluation, we have conducted a corpus study, analyzing the inferability of the generic methods in the collection libraries for Java, C#, Kotlin, and Ceylon. We found that, of the 724 generic methods across these libraries, 84% would be inferable in our $\text{Gen}_{\cap}$ calculus, and 95% would be inferable with some slight functional-preserving redesign. More details can be found in Appendix D.1. For our second evaluation, we designed a simple collection library specifically for our $\text{Gen}_{\cap}$ calculus. We were able to make most methods inferable, and we could even add more the functionality and stronger type-safety to the above existing collection libraries. Our collection library can be found in Appendix D.2.

$$
\begin{array}{lll}
\text{Expression} & e & ::= x \mid f(e,...) \mid \texttt{let}\ x := e\ \texttt{in}\ e \mid \texttt{throw} \\
& & \mid \texttt{if}\ e^g\ \texttt{then}\ e\ \texttt{else}\ e \mid \texttt{object}\ x : I\langle\vec{\tau}\rangle\ \{d;...\} \\
& & \mid e^m.m\langle\vec{\tau}\rangle(a;...) \mid \texttt{capture}\ e\ \texttt{as}\ x : I\langle\vec{\alpha}^!\rangle\ \texttt{in}\ e \\
& & \mid \texttt{capture}\ x\ \texttt{as}\ I\langle\vec{\alpha}^!\rangle\ \texttt{in}\ e \\
\text{Guard Expression} & e^g & ::= \alpha\ \texttt{satisfies}\ \sigma\ \texttt{as}\ \varsigma \mid \tau <: \alpha \mid \alpha <: \tau \\
& & \mid e\ \texttt{is}\ x : \tau \mid x\ \texttt{is}\ \tau \\
\text{Receiver Expression} & e^m & ::= e \mid e@\varsigma \\
\text{Program Argument} & a & ::= x \mapsto e \mid f(x,...) \mapsto e \\
\text{Capture Variables} & \vec{\alpha}^! & ::= \alpha^!,... \\
\text{Receiver Type} & \tau^m & ::= \tau \mid \sigma \\
\text{Method Definition} & d & ::= s \mapsto e \\
\text{Capture Variable} & \alpha^! & ::= \alpha \mid \_
\end{array}
$$

Method-Definition Typing $\Psi \mid \Theta^! \mid \Sigma \mid \Gamma \vdash d$

$$
\frac{\Psi \mid \Theta^! \mid \Sigma \vdash m\langle\Theta_e^!\rangle[\Theta_i](\Gamma_m)[\Sigma_m] : \tau \quad \vdash \langle\Theta_i\rangle \rightsquigarrow \langle\Theta_i^!\rangle \quad \Psi \mid \Theta^!, \Theta_e^!, \Theta_i^! \mid \Gamma, \Gamma_m \mid \Sigma, \Sigma_m \vdash e : \tau}{\Psi \mid \Theta^! \mid \Sigma \mid \Gamma \vdash m\langle\Theta_e^!\rangle[\Theta_i](\Gamma_m)[\Sigma_m] : \tau \mapsto e}
$$

Method-Invocation Typing $\Psi \mid \Theta^! \mid \Sigma \vdash \tau^m.m\langle\vec{\tau}\rangle(\Gamma) : \tau$

$$
\frac{}{\Psi \mid \Theta^! \mid \Sigma \vdash \bot.m\langle\bullet\rangle(\bullet) : \bot} \qquad \frac{}{\Psi \mid \Theta^! \mid \Sigma \vdash \bullet.m\langle\bullet\rangle(...; x : \bot; ...) : \bot}
$$

$$
\frac{\Psi \mid \Theta^! \mid \Sigma \vdash \tau_m.m\langle\vec{\tau}\rangle(\Gamma) : \tau \qquad \Psi \mid \Theta^! \mid \Sigma \vdash \tau'_m.m\langle\vec{\tau}\rangle(\Gamma) : \tau}{\Psi \mid \Theta^! \mid \Sigma \vdash (\tau_m \cup \tau'_m).m\langle\vec{\tau}\rangle(\Gamma) : \tau}
$$

$$
\frac{\Psi \mid \Theta^! \mid \Sigma \vdash \tau_m.m\langle\vec{\tau}\rangle(\Gamma) : \tau \qquad \Psi \mid \Theta^! \mid \Sigma \vdash \tau'_m.m\langle\vec{\tau}\rangle(\Gamma) : \tau'}{\Psi \mid \Theta^! \mid \Sigma \vdash (\tau_m \cap \tau'_m).m\langle\vec{\tau}\rangle(\Gamma) : \tau \cap \tau'}
$$

$$
\frac{\begin{array}{c}\Psi \vdash \tau^m.m\langle\Theta_m^!\rangle[\Theta_m](\Gamma_m)[\Sigma_m^\tau] : \tau \qquad \Psi \mid \Theta^! \vdash_? \langle\vec{\tau}\rangle : \langle\Theta_m^!\rangle \\ \Psi \mid \Theta^! \vdash_? \langle\vec{\tau}'\rangle : \langle\Theta_m\lfloor\vec{\tau}/\Theta_m^!\rfloor\rangle \qquad \Psi \mid \Theta^! \mid \Sigma \vdash \Sigma_m^\tau\lfloor\vec{\tau},\vec{\tau}'/\Theta_m^!,\Theta_m\rceil\end{array}}{\Psi \mid \Theta^! \mid \Sigma \vdash \tau^m.m\langle\vec{\tau}\rangle(\Gamma_m\lfloor\vec{\tau},\vec{\tau}'/\Theta_m^!,\Theta_m\rceil) : \tau\lfloor\vec{\tau},\vec{\tau}'/\Theta_m^!,\Theta_m\rceil}
$$

$$
\frac{\begin{array}{c}\Psi \mid \Theta^! \vdash \tau_m <: \tau'_m \\ \Psi \mid \Theta^! \vdash (\Gamma) <: (\Gamma') \qquad \Psi \mid \Theta^! \vdash \tau <: \tau' \qquad \Psi \mid \Theta^! \mid \Sigma \vdash \tau_m.m\langle\vec{\tau}\rangle(\Gamma') : \tau\end{array}}{\Psi \mid \Theta^! \mid \Sigma \vdash \tau'_m.m\langle\vec{\tau}\rangle(\Gamma) : \tau'}
$$

Figure 8.9: Expressions

Expression Typing $\Psi \mid \Theta^! \mid \Sigma \mid \Gamma \vdash e : \tau \qquad \Psi \mid \Theta^! \mid \Sigma \mid \Gamma \vdash e^m : \tau^m$

$$\frac{\Psi \mid \Theta^! \mid \Sigma \mid \Gamma \vdash e : \tau \qquad \Psi \mid \Theta^! \vdash \tau <: \tau'}{\Psi \mid \Theta \mid \Sigma \vdash e : \tau'} \qquad \frac{\varsigma : \alpha.\sigma \in \Sigma \qquad \Psi \mid \Theta^! \mid \Sigma \mid \Gamma \vdash e : \alpha}{\Psi \mid \Theta^! \mid \Sigma \mid \Gamma \vdash e@\varsigma : \sigma}$$

$$\frac{x : \tau \in \Gamma}{\Psi \mid \Theta^! \mid \Sigma \mid \Gamma \vdash x : \tau} \qquad \frac{f(\tau_1, ...) : \tau \in \Gamma \qquad \Psi \mid \Theta^! \mid \Sigma \mid \Gamma \vdash e_1 : \tau_1 \qquad ...}{\Psi \mid \Theta^! \mid \Sigma \mid \Gamma \vdash f(e_1, ...) : \tau}$$

$$\frac{\Psi \mid \Theta^! \mid \Sigma \mid \Gamma \vdash e_x : \tau_x \qquad \Psi \mid \Theta^! \mid \Sigma \mid \Gamma; x : \tau_x \vdash e : \tau}{\Psi \mid \Theta^! \mid \Sigma \mid \Gamma \vdash \textbf{let } x := e_x \textbf{ in } e : \tau}$$

$$\frac{}{\Psi \mid \Theta^! \mid \Sigma \mid \Gamma \vdash \textbf{throw} : \bot}$$

$$\frac{\begin{array}{c}\Psi \mid \Theta^! \mid \Sigma \mid \Gamma \vdash e^g : \Theta^!_g \mid \Sigma_g \mid \Gamma_g \\ \Psi \mid \Theta^!_g \mid \Sigma_g \mid \Gamma_g \vdash e_t \qquad \Psi \mid \Theta^! \mid \Sigma \mid \Gamma \vdash e_f : \tau\end{array}}{\Psi \mid \Theta^! \mid \Sigma \mid \Gamma \vdash \textbf{if } e^g \textbf{ then } e_t \textbf{ else } e_f : \tau}$$

$$\frac{\begin{array}{c}\Psi \mid \Theta^! \vdash_? I\langle\vec{\tau}\rangle \\ \Psi \mid \Theta^! \mid \Sigma \mid \Gamma; x : I\langle\vec{\tau}\rangle \vdash s_1 \mapsto e_1 \qquad ... \qquad \Psi \mid \Theta^! \mid \Sigma \vdash \{s_1; ...\} \sqsubseteq I\langle\vec{\tau}\rangle\end{array}}{\Psi \mid \Theta^! \mid \Gamma \mid \Sigma \vdash \textbf{object } x : I\langle\vec{\tau}\rangle \ \{s_1 \mapsto e_1; ...\} : I\langle\vec{\tau}\rangle}$$

$$\frac{\begin{array}{cc}\Psi \mid \Theta^! \vdash_? \tau_1 \qquad ... & \Psi \mid \Theta^! \mid \Sigma \mid \Gamma \vdash (a_1) : (p_1) \qquad ... \\ \Psi \mid \Theta^! \mid \Sigma \mid \Gamma \vdash e^m : \tau^m & \Psi \mid \Theta^! \mid \Sigma \vdash \tau^m.m\langle\tau_1, ...\rangle(p_1; ...) : \tau\end{array}}{\Psi \mid \Theta^! \mid \Sigma \mid \Gamma \vdash e^m.m\langle\tau_1, ...\rangle(a_1; ...) : \tau}$$

$$\frac{\begin{array}{c}\Psi \mid \Theta^! \mid \Sigma \mid \Gamma \vdash e : I\langle\vec{\tau}^a\rangle \\ \vdash \langle\vec{\alpha}^!\rangle := \langle\vec{\tau}^a\rangle \rightsquigarrow \langle\vec{\tau}\rangle \mid \Theta^!_\alpha \qquad \Psi \mid \Theta^!, \Theta^!_\alpha \mid \Sigma \mid \Gamma; x : I\langle\vec{\tau}\rangle \vdash e_x : \tau\end{array}}{\Psi \mid \Theta^! \mid \Sigma \mid \Gamma \vdash \textbf{capture } e \textbf{ as } x : I\langle\vec{\alpha}^!\rangle \textbf{ in } e_x : \tau}$$

$$\frac{\begin{array}{c}\Psi \mid \Theta^! \vdash \tau <: I\langle\vec{\tau}^a\rangle \\ \vdash \langle\vec{\alpha}^!\rangle := \langle\vec{\tau}^a\rangle \rightsquigarrow \langle\vec{\tau}\rangle \mid \Theta^!_\alpha \qquad \Psi \mid \Theta^!, \Theta^!_\alpha \mid \Sigma \mid \Gamma_1; x : \tau \cap I\langle\vec{\tau}\rangle; \Gamma_2 \vdash e_x : \tau\end{array}}{\Psi \mid \Theta^! \mid \Sigma \mid \Gamma_1; x : \tau; \Gamma_2 \vdash \textbf{capture } x \textbf{ as } I\langle\vec{\alpha}^!\rangle \textbf{ in } e_x : \tau}$$

$$\frac{\Psi \mid \Theta^! \mid \Sigma \mid \Gamma \vdash e : \tau_1 \qquad \Psi \mid \Theta^! \mid \Sigma \mid \Gamma \vdash e : \tau_2}{\Psi \mid \Theta^! \mid \Sigma \mid \Gamma \vdash e : \tau_1 \cap \tau_2}$$

Figure 8.9 (contd.): Expressions (Generics)

Guard-Expression Typing $\Psi \mid \Theta^! \mid \Sigma \mid \Gamma \vdash e^g : \Theta^! \mid \Sigma \mid \Gamma$

$$\frac{\Psi \mid \Theta^! \mid \Sigma \vdash [\varsigma : \alpha.\sigma]}{\Psi \mid \Theta^! \mid \Sigma \mid \Gamma \vdash \alpha \ \textbf{satisfies} \ \sigma \ \textbf{as} \ \varsigma : \Theta^! \mid \Sigma, \varsigma : \alpha.\sigma \mid \Gamma}$$

$$\frac{\Theta^! = \Theta^!_1, \tau_\ell <: !\alpha <: \tau_u, \Theta^!_2 \qquad \Psi \mid \Theta^!_1 \vdash_? \tau \qquad \Psi \mid \Theta^!_1 \vdash \tau_\ell <: \tau \qquad \Psi \mid \Theta^!_1 \vdash \tau <: \tau_u \qquad \Psi \mid \Theta^!_1, \tau <: !\alpha <: \tau_u, \Theta^!_2 \mid \varnothing \vdash [\Sigma]}{\Psi \mid \Theta^! \mid \Sigma \mid \Gamma \vdash \tau <: \alpha : \Theta^!_1, \tau <: !\alpha <: \tau_u, \Theta^!_2 \mid \Sigma \mid \Gamma}$$

$$\frac{\Theta^! = \Theta^!_1, \tau_\ell <: !\alpha <: \tau_u, \Theta^!_2 \qquad \Psi \mid \Theta^!_1 \vdash_? \tau \qquad \Psi \mid \Theta^!_1 \vdash \tau_\ell <: \tau \qquad \Psi \mid \Theta^!_1 \vdash \tau <: \tau_u \qquad \Psi \mid \Theta^!_1, \tau_\ell <: !\alpha <: \tau, \Theta^!_2 \mid \varnothing \vdash [\Sigma]}{\Psi \mid \Theta^! \mid \Sigma \mid \Gamma \vdash \alpha <: \tau : \Theta^!_1, \tau_\ell <: !\alpha <: \tau, \Theta^!_2 \mid \Sigma \mid \Gamma}$$

$$\frac{\Psi \mid \Theta^! \mid \Sigma \mid \Gamma \vdash e : \top}{\Psi \mid \Theta^! \mid \Sigma \mid \Gamma \vdash e \ \textbf{is} \ x : \tau : \Theta^! \mid \Sigma \mid \Gamma; x : \tau}$$

$$\frac{}{\Psi \mid \Theta^! \mid \Sigma \mid \Gamma_1; x : \tau'; \Gamma_2 \vdash x \ \textbf{is} \ \tau : \Theta^! \mid \Sigma \mid \Gamma_1; x : \tau' \cap \tau; \Gamma_2}$$

Method Lookup $\Psi \vdash \tau^m.s^\tau$

$$\frac{\textbf{interface} \ I\langle\Theta_I\rangle \ \textbf{extends} \ \bullet \ \textbf{satisfies} \ \bullet \ \{\ldots; s; \ldots\} \in \Psi}{\Psi \vdash I\langle\vec{\tau}\rangle.s\lfloor^{\vec{\tau}}/_{\Theta_I}\rceil}$$

$$\frac{\textbf{shape} \ S\langle\Theta_I\rangle \ \textbf{extends} \ \bullet \ \{\ldots; s; \ldots\} \in \Psi}{\Psi \vdash S\langle\vec{\tau}\rangle.s\lfloor^{\vec{\tau}}/_{\Theta_S}\rceil}$$

Program-Argument Typing $\Psi \mid \Theta^! \mid \Sigma \mid \Gamma \vdash (a) : (p)$

$$\frac{\Psi \mid \Theta^! \mid \Sigma \mid \Gamma \vdash e : \tau}{\Psi \mid \Theta^! \mid \Sigma \mid \Gamma \vdash (x \mapsto e) : (x : \tau)}$$

$$\frac{\Psi \mid \Theta^! \mid \Sigma \mid \Gamma; x_1 : \tau_1; \ldots \vdash e : \tau}{\Psi \mid \Theta^! \mid \Sigma \mid \Gamma \vdash (f(x_1, \ldots) \mapsto e) : (f(\tau_1, \ldots) : \tau)}$$

Figure 8.9 (contd.): Expressions (Generics)

8.6 EXPRESSIONS AND TYPE-CHECKING

At last, with the hierarchy in place and completely validated, we can finally discuss expressions, i.e. the only part of the program that actually executes. The grammar for our expressions is shown in Figure 8.9. There are of course variables and calls to higher-order parameters. Because our calculi have principal types, local (immutable) variables do not need any type annotations. Our calculi have exceptions, not because they are particularly interesting semantically here, but because they necessitate the $\bot$ type (for principal typing) and so impose an important constraint on the design of the type system.

An object in our calculi is an implementation of an interface that simply provides definitions of the necessary methods. These definitions can access variables in the context, making objects effectively closures, as well as the specified variable x representing the object itself. Checking whether the method definitions provide sufficient functionality for the declared interface is done just as with with interface inheritance. Note that, even if an object provides more functionality than is needed by its interface, its run-time type is its declared interface. Also note that objects know the type arguments for their run-time type, thereby providing reified generics.

One unusual feature of our calculi is the ability to explicitly **capture** the **in out** type arguments for a specific value as *capture* variables $\vec{\alpha}^!$. This is primarily provided to support use-site variance, as needed by Gen_α for joins, but it also illustrates some nice properties of the calculi. For one, the bounds on the capture variables can be inferred from the type of the expression due to principal typing, and the resulting kind context

and captured type are well-formed, ensuring that important invariants for decidability are maintained. Similarly, even if the principal type of the expression *inside* the **capture** references the capture variables, the variance requirements on bounds to interface type parameters ensure that there is a best way to approximate that principal type without reference to the capture variables. In other words, when the capture variables go out of scope, our calculus can automatically determine the best way to update the type of the expression appropriately.

The reason for the two calculi have different syntax for capturing is that $\text{Gen}_{\cap}$ can use intersection types to refine the type of the program variable x in the context, thereby providing flow-sensitive typing or occurrence typing [Komondoor et al., 2005; Tobin-Hochstadt and Felleisen, 2008], whereas Gen_{α} needs to introduce a new program variable with the captured type. This is also why they have different syntax for casting a value. Guards in our calculi can extend the kind, shape, and program contexts for the **then** branch to reflect the successful satisfaction, subtyping, or inhabitation check.

8.6.1 *Method Invocation*

Lastly, a method invocation specifies a receiver, the explicit type arguments, and the program arguments. Importantly, an invocation does not specify the inferable type arguments or the (inferable) shape evidence. A receiver is an expression or, in the case of Gen_{α}, an expression annotated with an evidence variable ς. In general, a type might be guaranteed to provide methods through the interfaces it implements *or* through the shapes it

satisfied. Sometimes a specific method can be provided through multiple means, and as such Gen_α uses evidence variables ς to prevent ambiguities about which means is intended by the programmer.

$\text{Gen}_\cap$ sidesteps this problem through its intersection expression-typing rule shown in blue in Figure 8.9. This rule indicates that if an expression can be given multiple types, then it can be given the intersection of those types. For many languages, this rule would be unsound because different typing proofs could result in different semantics, but the semantics of our calculi are coherent and therefore soundly model this rule. From a decidability perspective, this rule ensures principal types *provided* there a finite number of types that cover all of the possibilities. Furthermore, one needs to be able to determine what this finite cover is. In $\text{Gen}_\cap$, the only points of variability come from shape evidence (which there is a finite amount of) and (finite) intersection types, and the only points where this variability can cause complications is method invocation and capturing. Furthermore, principal-instantiation inheritance ensures that many intersection types are equivalent to some non-intersection type, and as such this rule is only needed in easy-to-recognize corner cases.

The rules for method-invocation typing are worth examining. Note that these rules are written declaratively, not algorithmically, and as such assume the argument types Γ have already been determined. The first rule addresses the case where the receiver has type $\bot$ and recognizes that this is unreachable (since $\bot$ is uninhabited in our calculi) by simply returning $\bot$. The second rule does the same but for first-order arguments, which is important because the type of that type might be used to uniquely determine an invariant type parameter, and $\bot$ is the only type that does not do so. The justification for the remaining rules should be straight-

forward from the discussion up to this point. The omitted rules for the judgement $\Psi \vdash \tau^m.m\langle\Theta^!\rangle[\Theta_m](\Gamma_m)[\Sigma^\tau_m] : \tau$ simply look up the method signature declared by the interface or shape of the receiving type.

8.6.2 *Decidable Type-Checking*

Despite the fact that the typing rules for our calculi have been specified declaratively, type-checking is in fact decidable. The algorithm is large due to the sheer complexity of the calculi, but the high-level reasoning is straightforward. First we ensure that subtyping is decidable, then we ensure the subshaping is decidable, then we ensure that shape satisfaction is decidable, then we ensure that method signatures or principally inferable, then we ensure methods are inherited appropriately, and finally we ensure that type-checking is unambiguous and locally inferable. Although we have not emphasized it, local type inference is important for inferring the return types of higher-order arguments from their inferred parameter types, which subsequently constrain the type arguments to a method. Thus, altogether type-checking, and indeed program validity itself, is decidable.

Theorem 8.1. *For all hierarchies Ψ and expressions e, the judgement $\vdash \Psi; e$ is decidable.*

8.7 SEMANTICS AND COHERENCE

With the exception of method invocation, the semantics of the calculi are overall unsurprising. They are, however, a little tedious to formalize. The

$$
\begin{array}{lll}
\text{Term} & t & ::= x \mid f(t,...) \mid \textbf{let } x := t \textbf{ in } t \mid \textbf{throw} \\
& & \mid \textbf{if } t^g \textbf{ then } t \textbf{ else } t \mid \textbf{object } x : I\langle\vec{\tau}\rangle\ \{d^t;...\} \\
& & \mid t^m.m\langle\vec{\tau}\rangle(a^t;...) \mid \textbf{capture } t \textbf{ as } x : I\langle\vec{\alpha}^!\rangle \textbf{ in } t \\
& & \mid \textbf{capture } x \textbf{ as } I\langle\vec{\alpha}^!\rangle \textbf{ in } t \mid \textbf{capture } v \textbf{ as } I\langle\vec{\alpha}^!\rangle \textbf{ in } t \\
\text{Guard Term} & t^g & ::= \tau \textbf{ satisfies } \sigma \textbf{ as } \varsigma \mid \tau <: \tau \mid t \textbf{ is } x : \tau \\
& & \mid x \textbf{ is } \tau \mid v \textbf{ is } \tau \\
\text{Receiver Term} & t^m & ::= t \mid t@\varsigma \\
\text{Argument Term} & a^t & ::= x \mapsto t \mid f(x,...) \mapsto t \\
\text{Argument Value} & a^v & ::= x \mapsto v \mid f(x,...) \mapsto t \\
\text{Term Context} & E & ::= f(v,...,E,t,...) \mid \textbf{let } x := E \textbf{ in } t \\
& & \mid \textbf{if } E \textbf{ is } x : \tau \textbf{ then } t \textbf{ else } t \mid E.m\langle\vec{\tau}\rangle(a^t;...) \\
& & \mid v.m\langle\vec{\tau}\rangle(a^v;...;x \mapsto E;a^t;...) \\
& & \mid \textbf{capture } E \textbf{ as } x : I\langle\alpha,...\rangle \textbf{ in } t \mid \odot \\
\text{Method Term} & d^t & ::= s^\tau \mapsto t \\
\text{Method Premise} & s^\tau & ::= m\langle\Theta^!\rangle[\Theta](\Gamma)(\Sigma^\tau) \\
\text{Value} & v & ::= \textbf{object } x : I\langle\vec{\tau}\rangle\ \{d^t;...\}
\end{array}
$$

Guard Reduction $\Psi \vdash t^g \to \top \quad \Psi \vdash t^g \to \bot$

$$\frac{\neg\, \Psi \mid \varnothing \mid \varnothing \vdash \tau.\sigma}{\Psi \vdash \tau \textbf{ satisfies } \sigma \textbf{ as } \varsigma \to \bot} \qquad \frac{\Psi \mid \varnothing \vdash \tau <: \tau'}{\Psi \vdash \tau <: \tau' \to \top} \qquad \frac{\neg\, \Psi \mid \varnothing \vdash \tau <: \tau'}{\Psi \vdash \tau <: \tau' \to \bot}$$

$$\frac{\neg\, \Psi \mid \varnothing \vdash I\langle\vec{\tau}\rangle <: \tau}{\Psi \vdash \textbf{object } \bullet : I\langle\vec{\tau}\rangle\ \{\bullet\} \textbf{ is } x : \tau \to \bot}$$

$$\frac{\Psi \mid \varnothing \vdash I\langle\vec{\tau}\rangle <: \tau}{\Psi \vdash \textbf{object } \bullet : I\langle\vec{\tau}\rangle\ \{\bullet\} \textbf{ is } \tau \to \top} \qquad \frac{\neg\, \Psi \mid \varnothing \vdash I\langle\vec{\tau}\rangle <: \tau}{\Psi \vdash \textbf{object } \bullet : I\langle\vec{\tau}\rangle\ \{\bullet\} \textbf{ is } \tau \to \bot}$$

Capture Reduction $\vdash \langle\vec{\alpha}^!\rangle := \langle\vec{\tau}\rangle \to \langle\vec{\tau}\rangle : \langle\Theta^!\rangle$

$$\frac{}{\vdash \langle\rangle := \langle\rangle \to \langle\rangle : \langle\rangle} \qquad \frac{\vdash \langle\vec{\alpha}^!\rangle := \langle\vec{\tau}\rangle \to \langle\vec{\tau}'\rangle : \langle\Theta^!\rangle}{\vdash \langle\vec{\alpha}^!, \_\rangle := \langle\vec{\tau}, \tau\rangle \to \langle\vec{\tau}'\rangle : \langle\Theta^!\rangle}$$

$$\frac{\vdash \langle\vec{\alpha}^!\rangle := \langle\vec{\tau}\rangle \to \langle\vec{\tau}'\rangle : \langle\Theta^!\rangle}{\vdash \langle\vec{\alpha}^!, \alpha\rangle := \langle\vec{\tau}, \tau\rangle \to \langle\vec{\tau}', \tau\rangle : \langle\Theta^!, \bot <: !\alpha <: \top\rangle}$$

Figure 8.10: Semantics

Term Reduction $\Psi \vdash t \to t$

$$\frac{\Psi \vdash t \to t'}{\Psi \vdash E[t] \to E[t']} \qquad \frac{}{\Psi \vdash E[\mathbf{throw}] \to \mathbf{throw}}$$

$$\frac{}{\Psi \vdash \mathbf{let}\ x := v\ \mathbf{in}\ t \to t\lfloor^{x \mapsto v}/_{x:\tau}\rceil}$$

$$\frac{\Psi \vdash t^g \to \top}{\Psi \vdash \mathbf{if}\ t_g\ \mathbf{then}\ t_t\ \mathbf{else}\ t_f \to t_t} \qquad \frac{\Psi \vdash t^g \to \bot}{\Psi \vdash \mathbf{if}\ t_g\ \mathbf{then}\ t_t\ \mathbf{else}\ t_f \to t_f}$$

$$\frac{v = \mathbf{object}\ \bullet : I\langle\vec{\tau}\rangle\ \{\bullet\} \qquad \Psi \mid \varnothing \vdash I\langle\vec{\tau}\rangle <: \tau}{\Psi \vdash \mathbf{if}\ v\ \mathbf{is}\ x : \tau\ \mathbf{then}\ t_t\ \mathbf{else}\ t_f \to t_t\lfloor^{x \mapsto v}/_{x:\tau}\rceil}$$

$$\frac{\Psi \mid \varnothing \mid \varnothing \vdash \tau.\sigma}{\Psi \vdash \mathbf{if}\ \tau\ \mathbf{satisfies}\ \sigma\ \mathbf{as}\ \varsigma\ \mathbf{then}\ t_t\ \mathbf{else}\ t_f \to t_t\lfloor\varsigma : \tau.\sigma\rceil}$$

$$\frac{\begin{array}{c} v = \mathbf{object}\ x : I\langle\vec{\tau}_v\rangle\ \{\ldots; m\langle\Theta^!\rangle[\Theta](p_1, \ldots)[\Sigma^\tau] : \bullet \mapsto t; \ldots\} \\ \Psi \mid \varnothing \vdash_? \langle\vec{\tau}'\rangle : \langle\Theta\lfloor^{\vec{\tau}}/_{\Theta^!}\rceil\rangle \qquad \Psi \mid \varnothing \mid \varnothing \mid \varnothing \vdash (a_1^v) : (p_1\lfloor^{\vec{\tau},\vec{\tau}'}/_{\Theta^!,\Theta}\rceil) \\ \ldots \qquad \Psi \mid \varnothing \mid \varnothing \vdash \Sigma^\tau\lfloor^{\vec{\tau},\vec{\tau}'}/_{\Theta^!,\Theta}\rceil \end{array}}{\forall\vec{\tau}''.\ \left(\begin{array}{c} \Psi \mid \varnothing \vdash_? \langle\vec{\tau}''\rangle : \langle\Theta\lfloor^{\vec{\tau}}/_{\Theta^!}\rceil\rangle \\ \Psi \mid \varnothing \mid \varnothing \mid \varnothing \vdash (a_1^v) : (p_1\lfloor^{\vec{\tau},\vec{\tau}''}/_{\Theta^!,\Theta}\rceil) \\ \vdots \\ \Psi \mid \varnothing \mid \varnothing \vdash \Sigma^\tau\lfloor^{\vec{\tau},\vec{\tau}''}/_{\Theta^!,\Theta}\rceil \end{array}\right) \implies \begin{array}{c} \Psi \mid \varnothing \\ \vdash \\ \langle\vec{\tau}'\rangle <: \langle\vec{\tau}''\rangle : \langle\Theta\rangle \end{array}}{\Psi \vdash v.m\langle\vec{\tau}\rangle(a_1^v; \ldots) \to t\lfloor^{\vec{\tau},\vec{\tau}'}/_{\Theta^!,\Theta}\rceil\lfloor^{x \mapsto v; a_1^v; \ldots}/_{x:I\langle\vec{\tau}_v\rangle; p_1; \ldots}\rceil\lfloor\Sigma^\tau\lfloor^{\vec{\tau},\vec{\tau}'}/_{\Theta^!,\Theta}\rceil\rceil}$$

$$\frac{\begin{array}{c} v = \mathbf{object}\ \bullet : I_v\langle\vec{\tau}_v\rangle\ \{\bullet\} \\ \Psi \mid \varnothing \vdash I_v\langle\vec{\tau}_v\rangle <: I\langle\vec{\tau}\rangle \qquad \vdash \langle\vec{\alpha}^!\rangle := \langle\vec{\tau}\rangle \to \langle\vec{\tau}'\rangle \mid \Theta^! \end{array}}{\Psi \vdash \mathbf{capture}\ v\ \mathbf{as}\ x : I\langle\vec{\alpha}^!\rangle\ \mathbf{in}\ t \to t\lfloor^{x \mapsto v}/_{x:I\langle\vec{\tau}\rangle}\rceil\lfloor^{\vec{\tau}'}/_{\Theta^!}\rceil}$$

$$\frac{\begin{array}{c} v = \mathbf{object}\ \bullet : I_v\langle\vec{\tau}_v\rangle\ \{\bullet\} \\ \Psi \mid \varnothing \vdash I_v\langle\vec{\tau}_v\rangle <: I\langle\vec{\tau}\rangle \qquad \vdash \langle\vec{\alpha}^!\rangle := \langle\vec{\tau}\rangle \to \langle\vec{\tau}'\rangle \mid \Theta^! \end{array}}{\Psi \vdash \mathbf{capture}\ v\ \mathbf{as}\ I\langle\vec{\alpha}^!\rangle\ \mathbf{in}\ t \to t\lfloor^{\vec{\tau}'}/_{\Theta^!}\rceil}$$

Figure 8.10 (contd.): Semantics (Generics)

t	$t\lfloor ... \rceil$ (where $\lfloor ... \rceil$ as shorthand for $\lfloor \vec{\tau}/\Theta \rceil \lfloor a_1^v; .../\Gamma \rceil \lfloor \Sigma^\tau \rceil$)
x	v where $x \mapsto v \in a_1^v; ...$
x	v where $\nexists v.\ x \mapsto v \in a_a^v; ...$
$f(t_1, ...)$	**let** $x_1 := t_1\lfloor ... \rceil$ **in** ... e where $f(x_1, ...) \mapsto e \in a_1^v; ...$
$f(t_1, ...)$	$f(t_1\lfloor ... \rceil, ...)$ where $\nexists x_1, ..., e.\ f(x_1, ...) \mapsto e \in a_1^v; ...$
throw	**throw**
if t^g **then** t_t **else** t_f	**if** $t^g\lfloor ... \rceil$ **then** $t_t\lfloor ... \rceil$ **else** $t_f\lfloor ... \rceil$
object $x : I\langle \vec{\tau}_x \rangle$ $\{d_1^t; ...\}$	**object** $x : I\langle \vec{\tau}_x \lfloor \vec{\tau}/\Theta \rceil \rangle$ $\{d_1^t\lfloor ... \rceil; ...\}$
$t^m.m\langle \vec{\tau} \rangle(a_1^t; ...)$	$t^m\lfloor \vec{\tau}/\Theta \rceil \lfloor a_1^v; .../\Gamma \rceil .m\langle \vec{\tau}\lfloor \vec{\tau}/\Theta \rceil \rangle (a_1^t\lfloor ... \rceil; ...)$
capture t **as** $x : I\langle \vec{\alpha}^! \rangle$ **in** t_x	**capture** $t\lfloor ... \rceil$ **as** $x : I\langle \vec{\alpha}^! \rangle$ **in** $t_x\lfloor ... \rceil$
capture x **as** $I\langle \vec{\alpha}^! \rangle$ **in** t	**capture** $x\lfloor ... \rceil$ **as** $I\langle \vec{\alpha}^! \rangle$ **in** $t\lfloor ... \rceil$
capture v **as** $I\langle \vec{\alpha}^! \rangle$ **in** t	**capture** $v\lfloor ... \rceil$ **as** $I\langle \vec{\alpha}^! \rangle$ **in** $t\lfloor ... \rceil$
t^g	$t^g\lfloor ... \rceil$ (where $\lfloor ... \rceil$ as shorthand for $\lfloor \vec{\tau}/\Theta \rceil \lfloor a_1^v; .../\Gamma \rceil \lfloor \Sigma^\tau \rceil$)
τ **satisfies** σ **as** ς	$\tau\lfloor \vec{\tau}/\Theta \rceil$ **satisfies** $\sigma\lfloor \vec{\tau}/\Theta \rceil$ **as** ς
$\tau_\ell <: \tau_u$	$\tau_\ell\lfloor \vec{\tau}/\Theta \rceil <: \tau_u\lfloor \vec{\tau}/\Theta \rceil$
t **is** $x : \tau$	$t\lfloor ... \rceil$ **is** $x : \tau\lfloor \vec{\tau}/\Theta \rceil$
x **is** τ	$x\lfloor ... \rceil$ **is** $\tau\lfloor \vec{\tau}/\Theta \rceil$
v **is** τ	$v\lfloor ... \rceil$ **is** $\tau\lfloor \vec{\tau}/\Theta \rceil$
t^m	$t^m\lfloor ... \rceil$ (where $\lfloor ... \rceil$ as shorthand for $\lfloor \vec{\tau}/\Theta \rceil \lfloor a_1^v; .../\Gamma \rceil \lfloor \Sigma^\tau \rceil$)
$t@\varsigma$	$t\lfloor ... \rceil$ where $\varsigma : \tau.\sigma \in \Sigma^\tau$
$t@\varsigma$	$t\lfloor ... \rceil@\varsigma$ where $\nexists \tau.\sigma.\ \varsigma : \tau.\sigma \in \Sigma^\tau$
a^t	$a^t\lfloor ... \rceil$ (where $\lfloor ... \rceil$ as shorthand for $\lfloor \vec{\tau}/\Theta \rceil \lfloor a_1^v; .../\Gamma \rceil \lfloor \Sigma^\tau \rceil$)
$x \mapsto t$	$x \mapsto t\lfloor ... \rceil$
$f(x_1, ...) \mapsto t$	$f(x_1, ...) \mapsto t\lfloor ... \rceil$
d^t	$d^t\lfloor ... \rceil$ (where $\lfloor ... \rceil$ as shorthand for $\lfloor \vec{\tau}/\Theta \rceil \lfloor a_1^v; .../\Gamma \rceil \lfloor \Sigma^\tau \rceil$)
$s^\tau \mapsto t$	$s^\tau\lfloor \vec{\tau}/\Theta \rceil \mapsto t\lfloor ... \rceil$
s^τ	$s^\tau\lfloor \vec{\tau}/\Theta \rceil$
$m\langle \Theta^! \rangle[\Theta'](\Gamma)[\Sigma^\tau] : \tau$	$m\langle \Theta^!\lfloor \vec{\tau}/\Theta \rceil \rangle[\Theta'\lfloor \vec{\tau}/\Theta \rceil](\Gamma\lfloor \vec{\tau}/\Theta \rceil)[\Sigma^\tau\lfloor \vec{\tau}/\Theta \rceil] : \tau\lfloor \vec{\tau}/\Theta \rceil$
Σ^τ	$\Sigma^\tau\lfloor \vec{\tau}/\Theta \rceil$
$\varsigma_1 : \tau_1.\sigma_1, ...$	$\varsigma_1 : \tau_1\lfloor \vec{\tau}/\Theta \rceil.\sigma_1\lfloor \vec{\tau}/\Theta \rceil, ...$

Figure 8.11: Program-Argument Substitution

issue is that exceptions were designed to ensure decidability, but once the program has been type-checked, reducing the program can disrupt some of the structures in expressions that we relied upon for decidability. For example, the expression implementing a generic method can branch on whether or not a type *variable* satisfies a given shape, updating the shape context appropriately within the branch, but once that generic method gets invoked with actual type arguments then that type variable gets substituted with an actual type. Consequently, we introduce a new grammar of terms t, defined in Figure Figure 8.10, that contains the grammar of expressions by generalizing many uses of type variables to arbitrary types. Similarly, because the typing rules for expressions were designed to ensure both soundness *and* decidability, we introduce a more relaxed typing judgement $\Psi \mid \Theta \mid \Sigma^{\tau} \mid \Gamma \vdash^{t} t$ for terms that only ensures soundness and inferability of methods in objects (for reasons that will become clear). These rules are straightforward adaptions of the rules for expressions, so we defer them to Appendix D.4.

8.7.1 *Method Invocation*

The only oddity of our semantics is method invocation. Recall `singleton` from the beginning of the chapter. At run time in C#, it returns a `List` whose invariant type argument depends on the inferred type argument to singleton, for which there are often multiple valid options. This means whether later casts of the returned value will succeed or fail can depend on the result of type-argument inference. For gradualizability, we want to ensure semantic coherence, and $\text{Gen}_{\cap}$ even relies on semantic coherence

in order to achieve decidability via its intersection rule. Thus we cannot use C#'s semantics for generic methods, and in fact we had to develop a semantics for generic methods that to our knowledge is entirely new.

The biggest of the term reduction rules in Figure 8.10 is easier to describe in words than with formulae. It says, using the *object*'s specification of the method, to infer the *principal* instantiation of the type arguments (according to their declared variances) for which the *run-time* types of the program arguments are considered valid argument types and for which the necessary shape evidence can be constructed. Any references to the inferred type parameters within the expression implementing the method are then substituted with these inferred type arguments. Notice that the semantics refers to only run-time aspects of the invocation, meaning that (unlike C#) the type arguments that were inferred at compile time are semantically meaningless—they just ensured that the invocation was safe and determined an upper bound on the run-time type of the returned value.

One might worry that this semantics would be prohibitively expensive, but frequently type parameters are not used in any way that is visible at run time. In these cases, our semantics is in fact more efficient because we are not constructing or passing around unused run-time arguments. Even when they are used, often the type arguments that are inferred at run time are simpler to construct than those at compile time because they are developed with complete information whereas at compile time they are more likely to be relying on constructs such as union and intersection types in order to increase the polymorphic reusability of generic methods. All in all, one could imagine either semantics performing better than

the other, varying by specific patterns in a code base, and it would be interesting future work to experimentally evaluate their relative behaviors.

Lastly, as an anecdote, some years ago the Ceylon team encountered usability problems with the standard reification semantics. Users were expecting the pair value `["Hello", 5]` to have run-time type `Pair⟨String, Integer⟩` and were surprised to discover it could have run-time type `Pair⟨Object, Object⟩` depending on what was known of the types of the contained values at *compile* time. Consequently, the Ceylon team changed their semantics for tuples to dynamically determine their type arguments based on their contained values. Since the type parameters for `Pair` are covariantly inferable, this change actually coincides with our semantics.

8.7.2 *Progress, Preservation, and Semantic Coherence*

Our semantics for method invocation is specified declaratively. That is, our semantics does not specify an algorithm to use to compute the principal type arguments, if one even exists. As such, one might worry that well-typed programs might not reduce reliably. This issue is addressed by the fact that method signatures, even of object terms, must satisfy the previously discussed method-signature-inferability judgement, which guarantees that principal type arguments always exist if any exist (which in turn is guaranteed by typability of the term). Furthermore, a proof of this judgement provides an algorithm to use to infer the principal type arguments.

Theorem 8.2 (Progress). *For all hierarchies* Ψ *and terms* t*, if* $\vdash^t \Psi; t$ *holds then either* t *is a value* v*, or* t *is an exception* **`throw`**, *or there exists a computable* t' *such that* $\Psi \vdash t \to t'$ *holds.*

One might worry that different proofs of inferability might provide different algorithms. While technically true, the differences are simply due to the fact that equivalent types can be syntactically different. The judgement $\Psi \vdash t \approx t$, formalized in Figure D.5, indicates that two terms are equivalent modulo equivalence of types. Because principal type arguments are unique up to equivalence, our semantics reduces equivalent terms to equivalent terms.

Theorem 8.3 (Coherence). *For all hierarchies* Ψ *and terms* t_1*,* t_2*,* t_1'*, and* t_2'*,*

$$\Psi \vdash t_1 \approx t_2 \quad \wedge \quad \Psi \vdash t_1 \to t_1' \quad \wedge \quad \Psi \vdash t_2 \to t_2' \quad \Longrightarrow \quad \Psi \vdash t_1' \approx t_2'$$

This, in combination with the fact that our semantics is not specified using any form of type-directed translation, guarantees that our semantics is coherent. In the case of $\text{Gen}_\cap$, coherence is necessary for ensuring the soundness of its intersection rule.

Theorem 8.4 (Preservation). *For all hierarchies* Ψ *and terms* t *and* t'*,*

$$\vdash^t \Psi; t \quad \wedge \quad \Psi \vdash t \to t' \quad \Longrightarrow \quad \vdash^t \Psi; t'$$

8.8 GRADUALIZABILITY

Type-argument inference is straightforwardly gradualizable for first-order parameters in systems where every value knows it most precise type, as in

Nom in Chapter 5. However, there is a caveat with respect to higher-order parameters: if the lambda given as a higher-order argument has a dynamic return type, type arguments cannot be inferred from it. One way to address this may be to distinguish type arguments that are erasable from those that are not, and restrict higher-order parameters to only constrain erasable arguments, which can be ignored at run time anyway. The downside of this approach is that erasable arguments are both restricted in how they can be used and on the other hand might leak into other parts of the code. We leave this problem for future work.

8.9 SUMMARY

This chapter presented two calculi with decidable and largely gradualizable generics, and our corpus and case studies suggest that the restrictions these calculi (or at least $\text{Gen}_{\cap}$) impose are indeed practical. This was made possible by changing to a declaration-site, rather than use-site, perspective on type-argument inference, and by recognizing that there is a symbiotic relationship between decidability and gradualizability. Furthermore, prior works on subtyping and on the binary-method problem formed the foundation for establish the invariants our strategy heavily relies upon. With this infrastructure in place, we were able to provide many features that are critical to how generics are used in industry while also maintaining the principles advocated by the programming-languages research community. Thus this work takes a major step towards practical decidable and gradualizable generics, and yet there are still many more steps ahead.

9

EPILOGUE

The previous sections of this dissertation presented several major improvements on the state of the art in both gradual typing and type system design in general. Yet, there is still much to do. In the following, we give an overview of some particularly interesting challenges that lie ahead.

9.1 FUTURE WORK

9.1.1 *Generics*

As stated in Section 8.8, work on making generics practical is not quite done yet. Besides making type-argument inference work in the face of structural values and dynamic types in higher-order parameters, there is also the question of whether it is possible to use generic types "raw" and discover their type parameters later. One might imagine that programmers writing untyped code would like to use the standard generic type `List⟨T⟩` as just `List`, without having to specify the type argument immediately. However, if they proceed to only fill that list with values of type `String`, they might reasonably expect to be able to give this object to code that expects a `List⟨String⟩`. This feature is supported by most other gradually typed languages to date, but it is not immediately clear whether there is a way to support it both soundly and efficiently, as current (sound)

implementations either have to recursively check the whole structure or allocate wrappers. It might be possible to have the instance monotonically keep track of upper and lower bounds for its type variables based on how it is used (covariant uses push upper bounds down, contravariant uses push lower bounds up), and only raise an error if bounds go past each other. Under such a scheme, a `List` could be cast to `List⟨String⟩` if all previously added elements (representing contravariant uses) were subtypes of `String` and no other code demanded to be able to read some type from the list that is not a supertype of `String` (say by casting it to `Iterable⟨Integer⟩`). However an instance of `List` with undetermined type arguments would again not allow us to infer concrete instantiations for type variables in generic type-argument inference.

A possible solution is to also allow these arguments to stay uninstantiated for the moment and keep them around as unification variables, because if, say, a typed generic method creates an instance of `List⟨T⟩` based on an undetermined type argument T, any uses of the new instance of `List` and the instance of `List` coming from untyped code now affect both of them since their uses and instantiations of their type parameter need to stay consistent. These dependency chains might become quite long, but they might also be quite regular. More research and good benchmarks would be needed to establish if this approach can be practical.

9.1.2 *Branching based on run-time types*

Many languages allow a programmer to determine the type of a value at runtime and execute different code based on the result. For example,

programmers may write the following code to handle different kinds of values stored in variable x.

```
if(x instanceof A) { ... }
else { ... }
```

For any gradually-typed language with non-transparent, not immediately accountable casting, such as MonNom, this presents a problem: what if x *could* be an instance of A if we cast it, but does not have to? Both the then- and the else-branch are plausible options in that case, but in order to satisfy the gradual guarantee, we have to assume that there is a version of the program where a correct type annotation somewhere would force our hand here, and we have to magically make that same decision, which is impossible.

One way to address this issue might be to change the semantics of **if**/**instanceof** constructs such that the programmer has to specify a "preferred" branch, which the run-time will try to get into if it can by casting the value appropriately. The gradual guarantee would also have to be modified to allow divergences in the program result—more precise type information would lead to values produced by more preferred branches. This would be a significant change to the formulation of the gradual guarantee, as the relationship between the results would now also be based on how one obtained them, not just on what they are.

Explicit **instanceof**-checks are not the only construct with this problem. With gradual typing, overloaded methods may have to be resolved at run time in what is essentially an **instanceof**-based decision tree, where a value may again be compatible with multiple different overloadings of a method. Preferences between overloadings that might be defined at

points in the code that are distant from each other would be harder to conveniently specify, so overloading may have to be restricted further.

9.1.3 *Optimizations*

In Chapter 6 we purposefully omitted several well-known optimization techniques that might endager the validity of our small number of benchmarks. However, there are many more specialized optimization techniques that are worth exploring. The most interesting one would be to dynamically recompile the code of structural values that have been monotonically cast if it turns out that the now available type information can be used to optimize the code. This might not be useful for every piece of code, but many modern virtual machines already make decisions of what code to (re-)compile and optimize based on run-time statistics. One would expect that, in particular, many simple lambdas could benefit from such an optimization; that is, lambdas of the form $\lambda x.x$, $\lambda x.x + n$, $\lambda x.x.foo(5)$, and so on.

9.1.4 *Gradualizability*

Some authors have previously explored more or less automatically adding gradual typing to typed languages [Cimini and Siek, 2016, 2017; Garcia, Clark, and Tanter, 2016], along with other meta-properties of gradual typing [New and Ahmed, 2018; New, Licata, and Ahmed, 2019; Siek, Vitousek, Cimini, and Boyland, 2015; Toro, Labrada, and Tanter, 2019]. In this dissertation, we have shown that there are several important type-

system features for which this is simply not possible, at least not without substantially changing the design of the particular feature. It would be interesting to have a theory of *gradualizability* to inform us more generally about which type-system features are gradualizable and what makes them so.

9.2 CONCLUSION

There is still much to do before sound gradual typing can be added to a modern object-oriented programming language in a way that is usable in industry. However, the work in this dissertation makes us hopeful that sound gradual typing is not dead, and has a real chance of one day becoming an integral part of major general-purpose languages. This is by far not the only challenge, but it seems to be the biggest remaining one. In particular, we know that with careful design, making sure that the selected language features work well with each other and gradual typing and can be efficiently implemented, a gradually typed language can have the major theoretical properties that have been formalized so far *and* be efficient.

Part IV

APPENDICES

A

SHAPE ANALYSIS

Owner	Shape
drjava-r5756	Finalizable
drools-core	AbstractBaseLinkedListNode
drools-core	Entry
drools-core	LinkedList
drools-core	LinkedListNode
drools-core	TraitableBean
findbugs-1.3.9	AbstractEdge
findbugs-1.3.9	GraphEdge
findbugs-1.3.9	GraphVertex
findbugs-1.3.9	AbstractVertex
findbugs-1.3.9	AnnotationEnueration
hadoop-1.1.2	WritableComparable
informa-0.7.0-alpha2	WithChildrenMIF
jre-1.6.0	Comparable
jre-1.6.0	Enum
junit-4.10	FrameworkMember
pmd-4.2.5	MemberNode

Figure A.1: Table of all shapes found by our analysis

Project Name	JRE Version	LOC	# C	# I
android-async	1.7	2719	29	1
antlr-3.4	1.6.0	50625	93	13
AoIsrc281	1.5.0_22	112161	460	27
apache-ant-1.9.2	1.6.0	106072	1078	94
argouml	1.7.0	176428	2028	565
castor-1.3.3	1.6.0	235466	1648	151
checkstyle-src-5.1	1.5.0_22	61530	286	18
chunk-templates-2.4	1.7.0	12823	97	5
disruptor	1.7	3135	38	21
drjava-r5756	1.7	90753	848	95
drools-core	1.7	146139	1693	316
eclipse_SDK-3.7.1	1.6.0	2489153	20954	3459
findbugs-1.3.9	1.5.0_22	110932	1111	124
fitnesse	1.6.0	64459	1143	77
freecol	1.6.0	106412	724	32
freecs-1.3.20100406	1.5.0_22	24453	130	16
geogit-core	1.7.0	44553	436	24
geotools-9.2	1.6.0	961944	6794	1494
gitblit	1.7	54057	514	41
hadoop-1.1.2	1.7.0	313519	3522	183
heritrix-1.14.4	1.5.0_22	64916	599	51
informa-0.7.0-alpha2	1.5.0_22	13874	115	45
javacc-5.0	1.5.0_22	21299	209	8
java2objc	1.7.0	3720	86	0
jboss-5.1.0	1.7.0	557866	2588	495
jbrowser-1.9.1	1.7.0	11163	128	27
jEdit	1.5.0_22	109516	876	63
jfin	1.5.0_22	3999	45	2
jgrapht-0.8.1	1.6.0	17233	201	33
JGroups-2.10.0.GA.src	1.6.0	96404	933	80

Figure A.2: Table of projects, JRE version used in development, total lines of code (LOC), number of classes (C), and number of interfaces (I). Class and interface counts include nested declarations.

Project Name	JRE Version	LOC	# C	# I
joptsimple	1.7.0	1943	40	5
jzmq	1.7	931	17	0
jre-1.6.0	1.6.0	884435	7896	1700
jRuby-1.7.9	1.7.0	270537	1693	140
jspwiki-2.8.4	1.5.0_22	60250	392	36
junit-4.10	1.5.0_22	6850	137	11
lucene-3.5.0	1.6.0	296778	739	41
marauroa-3.8.1	1.5.0_22	17734	178	13
megamek-0.35.18	1.6.0	242836	1760	64
MyJDownloaderClient	1.7.0	3803	97 16	
netbeans-7.3	1.6.0	4631942	36105	4211
openjdk-7-fcs/langtools	1.7.0	93533	844	237
picasso	1.7	2781	79	8
picocontainer	1.5.0_22	9254	173	29
pmd-4.2.5	1.5.0_22	60062	792	52
poi-3.6	1.5.0_22	9254	2136	131
retrofit	1.7	7664	65	25
sablecc-3.7	1.6.0	29496	215	5
springframework-3.0.5	1.6.0	323008	2511	419
sprinkles	1.7	1261	24	5
struts-2.2.1	1.5.0_22	143311	1965	167
sunflow	1.5.0_22	21970	185	19
tapestry-core	1.5.0_22	67390	572	232
trove-2.1.0	1.5.0_22	5846	45	9
wct-1.5.2	1.6.0	245849	487	62
weka	1.6.0	52297	1383	148

Figure A.3: Table of projects, JRE version used in development, total lines of code (LOC), number of classes (C), and number of interfaces (I). Class and interface counts include nested declarations (contd.).

B

MORE ON NOM

B.1 INFERRING DISPATCH MODES

Dynamic dispatch is an important part of object-oriented programming languages. As such, we need it to be implemented efficiently in a well-behaved manner. For efficiency, the core challenge is devising a quick and uniform process for looking up a specific object's implementation of a method without knowing which of a wide variety of objects is being accessed. For behavior, the core challenge is that the caller of a method may view it differently than the implementor of the method due to features such as inheritance and subtyping. While these challenges need to be addressed by all object-oriented languages, gradual typing introduces new layers of complexity to each of them.

In our calculus, we demonstrate that our strategy for addressing these challenges is to use *dispatch modes*. This is really an extension of what is already done for nominally typed object-oriented languages. When the dispatch mode is a class, the mode indicates that we can and will look up a method's implementation by accessing a fixed offset within the object's virtual-method table, taking advantage of the fact that full Nom enforces single class inheritance. When the dispatch mode is an interface, the mode indicates that we can and will look up a method's implementation by first looking up the address of the appropriate interface-method table

within the object's interface table and then accessing a fixed offset within that interface-method table. These dispatch modes capture how dynamic dispatch is implemented in nominally typed object-oriented languages. To efficiently extend this approach to gradual typing, we introduce a **dyn** dispatch mode. When the dispatch mode is **dyn**, the mode indicates that we will look up a method's implementation by accessing the object's dynamic-dispatch hashtable, failing if no appropriate entry exists. This is how dynamic dispatch is implemented by dynamically typed object-oriented languages. Thus our dispatch modes combine the implementation strategies for both statically typed and dynamically typed languages.

In our calculus, these dispatch modes are already specified. However, in a real gradually typed object-oriented language, they would be inferred at compile time, as we do in Nom. We have already shown that our annotated language is well-behaved, so here we discuss how to make the unannotated language be similarly well-behaved.

B.1.1 *Restricting Dispatch Modes*

Observe that, in theory, one could always infer the **dyn** dispatch mode. However, this would mean the type-checker could never reject a method invocation. Conceptually, although **dyn** provides a way to make our type-checker more optimistic, we only want to rely on optimism when so directed by the programmer. Thus, if the receiver of a method invocation has a pessimistic type, i.e. a type besides **dyn**, then the invocation should be valid only if there exists a pessimistic dispatch mode for which the

receiver and arguments are optimistically acceptable. We call such an invocation *pessimistically dispatchable*.

B.1.2 *Resolving Ambiguities*

Consider the following hierarchy:

```
interface I1 {
    Integer foo1(Integer a1, dynamic a2);
}
class C1 implements I1 {
    Integer foo1(dynamic a1, Integer a2) {...}
}
class Sub1 extends C1 {
    Integer foo1(Number a1, Number a2) {...}
}
```

Suppose we have a call to `foo1` on a receiver expression of type `C1` and with two argument expressions each of type **`dyn`**. The question is what dispatch mode, if any, should be inferred for this invocation. First observe that it is pessimistically dispatchable, as evidenced by both the `C1` and `I1` dispatch modes. Thus, we should infer *some* dispatch mode for this invocation.

The question, then, is which dispatch mode should be inferred, since we have already seen that there are multiple optimistically typed candidates. The problem is that each dispatch mode imposes a different cast on the arguments. `I1` requires the *first* argument to be cast to `Integer`, whereas `C1` requires the *second* argument to be cast. Thus the choice of inferred

dispatch mode can affect the semantics of the program. We want the semantics to be easy to predict by a non-expert user, and we want the semantics to be stable under simple program transformations that most users would expect to be inconsequential (such as weakening the static type of the receiver to be just `I1`). So to rectify this problem, here we must turn to design tradeoffs.

One possible solution is to determine the dispatch mode at run time based on the the run-time types of the arguments. If the first argument is an `Integer`, then use the `I1` dispatch mode. If the second argument is an `Integer`, then use the `C1` dispatch mode. If both arguments are `Integer`s, then either dispatch mode can be chosen without affecting the semantics of that particular run-time invocation. If neither argument is an `Integer`, then indicate a run-time type failure, much like a failed cast or a failed dynamic method lookup. The advantage of this solution is that the run-time semantics precisely reflects the semantics of the statically typed language: the gradually typed expression executes just as if it were statically typed, but with its **`dyn`** components filled in with their run-time types. This is the approach implemented by C# [Bierman, Meijer, and Torgersen, 2010] and advocated for by Garcia, Clark, and Tanter [2016].

The disadvantage, though, is poor performance due to heavy branching on run-time type information. That is, it essentially amounts to deferring compilation of the method invocation until run time, despite the fact that everything we need to know to compile this method is available to us at compile-time except simply which casts to insert. Thus another possible solution is to simply use the **`dyn`** dispatch mode in such ambiguous cases. This way we can go straight to the dynamic dispatcher, which will simply redirect to the method's implementation along with precompiled

checks that the arguments are valid. Note that this shows that even when the receiver has a pessimistic type, it can still be appropriate to use the **`dyn`** dispatch mode. This solution is more efficient, but it does have a disadvantage. In particular, suppose the run-time type of the receiver was `Sub1`, and the run-time types of the arguments were both `Float`. Then this solution would allow execution to proceed even though neither of the statically available dispatch modes would accept those arguments.

It is worth noting that we encountered multiple situations forcing this design tradeoff between precise reflection of the statically typed language and efficient implementation of mixed-type code. The criteria discussed in this chapter, including the gradual guarantee, provide no insights into how to make this tradeoff. For Nom, we considered each situation separately, rather than always appealing to one extreme or the other. For example, with method overloading, which we do not discuss in the chapter, we allow an overriding of an overloading to proceed with more lax arguments, as above, but we prevent invocations from proceeding with an overloading that is unrelated to the overloadings that were statically available. We do this because we view unrelated overloadings as having possibly unrelated specifications, whereas we view overridden overloadings as having related specifications with possibly relaxed requirements. Thus our semantics guarantees that execution proceeds only with a statically available overriding specification, although possibly a variant with more relaxed requirements.

B.1.3 *Aggregating Return Types*

Now consider the following hierarchy:

```
interface I2 {
    Integer foo2();
}
class C2 implements I2 {
    dynamic foo2() {...}
}
```

Suppose we have a variable `x2` of type `C2`, and suppose we call `foo2` on `x2` and print the result to the console. This is pessimistically dispatchable, and neither dispatch mode imposes any casts on any arguments (since there are no arguments), so we can use either one. Virtual-method tables are more efficient, so we choose to use the `C2` dispatch mode.

Now suppose the implementation of `foo2` returns a `String`. According to the semantics of our calculus, this would succeed with the `C2` dispatch mode and the `String` would proceed to be printed to the console. However, suppose we make the seemingly harmless transformation of changing the declared type of `x2` to be just `I2` instead of `C2`. Now the inferred dispatch mode would be `I2`, whose implementation would cast the returned value to `Integer`, which would fail. Thus this seemingly innocuous change would in fact significantly change the semantics of the program. And interestingly, this is perfectly acceptable for the gradual guarantee, since we are replacing a static type with a different static type, not a dynamic type.

Nonetheless, we believe this would be unacceptable to many programmers. So to address this problem, Nom aggregates the return types of inherited method specifications. This can be implemented in the language specification either by requiring return types to be *pessimistic* subtypes of inherited return types, or by inserting casts to inherited return types into the method definition. The downside of the former implementation is that it occasionally requires untyped classes implementing typed interfaces to explicitly state a return type. The downside of the latter implementation is that the pessimistic semantics of untyped method implementations need to be revised to incorporate inherited return types, a layer of indirection that may not be obvious to programmers. Nom implements the latter so that untyped code can remain untouched. Overall, our experience with dispatch modes points at a non-trivial design space that is to be further explored.

B.2 PROOF OF SOUNDNESS

The use of evaluation contexts in our operational semantics often complicate proofs unnecessarily. In Figure B.1, we present formalizations of **`bad-cast`** and **`erroneous`** without evaluation contexts, as well as introduce a new judgement $\Psi \vdash e$ **`erroneous`** ε indicating that ε is the particular error indicated by e. These formalizations of **`bad-cast`** and **`erroneous`** are equivalent to the ones in Figure 5.6, so we abuse notation and denote both pairs of judgements in the same manner. In Figure B.2, we present a formalization of our reduction rules without evaluation contexts in Figure B.2.

Errors without Evaluation Contexts $\Psi \vdash e$ **erroneous**, $\Psi \vdash e$ **bad-cast**

$$\frac{\Psi \vdash e \textbf{ erroneous cast } v \textbf{ to } C}{\Psi \vdash e \textbf{ bad-cast}} \qquad \frac{\Psi \vdash e \textbf{ erroneous } \varepsilon}{\Psi \vdash e \textbf{ erroneous}}$$

$$\frac{\Psi \vdash_{\blacktriangleleft} v \qquad v = C(\dots) \qquad \neg\, \Psi \vdash C \blacktriangleleft C'}{\Psi \vdash \textbf{cast } v \textbf{ to } C' \textbf{ erroneous cast } v \textbf{ to } C'}$$

$$\frac{\Psi \vdash_{\blacktriangleleft} v \qquad v = C(\dots) \qquad \textbf{class } C(f^1, \dots) \in \Psi \qquad \nexists i.\ f = f^i}{\Psi \vdash v.f_{\textbf{dyn}} \textbf{ erroneous } v.f_{\textbf{dyn}}}$$

$$\frac{\begin{array}{c}\Psi \vdash_{\blacktriangleleft} v \\ \forall i.\ \Psi \vdash_{\blacktriangleleft} v_i \qquad v = C(\dots) \qquad \nexists \tau, \tau_1, \dots, \tau_n.\ \tau\ C.m(\tau_1, \dots, \tau_n) \in \Psi\end{array}}{\Psi \vdash v.m_{\textbf{dyn}}(v_1, \dots, v_n) \textbf{ erroneous } v.m_{\textbf{dyn}}(v_1, \dots, v_n)}$$

$$\frac{\Psi \vdash e \textbf{ erroneous } \varepsilon}{\Psi \vdash \textbf{let } \tau\ x := e \textbf{ in } e' \textbf{ erroneous } \varepsilon}$$

$$\frac{\textbf{class } C(\tau_1, \dots, \tau_n) \in \Psi \qquad \forall j.\ \Psi \vdash v_j \blacktriangleleft \tau_j \qquad \Psi \vdash e_{i+1} \textbf{ erroneous } \varepsilon}{\Psi \vdash C(v_1, \dots, v_i, e_{i+1}, \dots, e_n) \textbf{ erroneous } \varepsilon}$$

$$\frac{\Psi \vdash e \textbf{ erroneous } \varepsilon}{\Psi \vdash e.f_\delta \textbf{ erroneous } \varepsilon} \qquad \frac{\Psi \vdash e \textbf{ erroneous } \varepsilon}{\Psi \vdash e.m_\delta(e_1, \dots, e_n) \textbf{ erroneous } \varepsilon}$$

$$\frac{\begin{array}{c}\Psi \vdash \delta.m(\tau_1, \dots, \tau_n) : \tau \\ \Psi \vdash v \blacktriangleleft \delta \qquad \forall j.\ \Psi \vdash v_j \blacktriangleleft \tau_j \qquad \Psi \vdash e_{i+1} \textbf{ erroneous } \varepsilon\end{array}}{\Psi \vdash v.m_\delta(v_1, \dots, v_i, e_{i+1}, \dots, e_n) \textbf{ erroneous } \varepsilon}$$

$$\frac{\Psi \vdash e \textbf{ erroneous } \varepsilon}{\Psi \vdash \textbf{cast } e \textbf{ to } \tau \textbf{ erroneous } \varepsilon}$$

Figure B.1: Errors without Evaluation Contexts

Reduction Rules without Evaluation Contexts $e \; R \; e$

$$\frac{\bar{\Psi} \vdash e_x \; R \; e'_x}{\bar{\Psi} \vdash \texttt{let}\ \tau\ x \coloneqq e_x\ \texttt{in}\ e \; R \; \texttt{let}\ \tau\ x \coloneqq e'_x\ \texttt{in}\ e}$$

$$\frac{(\bar{\Psi} \vdash v \triangleleft \tau)}{\bar{\Psi} \vdash \texttt{let}\ \tau\ x \coloneqq v\ \texttt{in}\ e \; R \; e[x \mapsto v]}$$

$$\frac{\begin{array}{c}\bar{\Psi} \vdash e_{i+1} \; R \; e'_{i+1} \\ (\texttt{class}\ C(\tau_1, \ldots, \tau_n) \in \bar{\Psi}) \\ (\forall j.\ \bar{\Psi} \vdash v_j \triangleleft \tau_j)\end{array}}{\bar{\Psi} \vdash C(v_1, \ldots, v_i, e_{i+1}, \ldots, e_n) \; R \; C(v_1, \ldots, v_i, e'_{i+1}, \ldots, e_n)}$$

$$\frac{\bar{\Psi} \vdash e \; R \; e'}{\bar{\Psi} \vdash e.f_\delta \; R \; e'.f_\delta} \qquad \frac{v = C(v_1, \ldots) \qquad \texttt{class}\ C(f^1, \ldots) \in \bar{\Psi} \qquad (\bar{\Psi} \vdash v \triangleleft \delta)}{\bar{\Psi} \vdash v.f^i_\delta \; R \; v_i}$$

$$\frac{\bar{\Psi} \vdash e \; R \; e'}{\bar{\Psi} \vdash e.m_\delta(e_1, \ldots, e_n) \; R \; e'.m_\delta(e_1, \ldots, e_n)}$$

$$\frac{\begin{array}{c}\bar{\Psi} \vdash e_{i+1} \; R \; e'_{i+1} \\ (\bar{\Psi} \vdash \delta.m(\tau_1, \ldots, \tau_n) : \tau) \qquad (\bar{\Psi} \vdash v \triangleleft \delta) \qquad (\forall j.\ \bar{\Psi} \vdash v_j \triangleleft \tau_j)\end{array}}{\bar{\Psi} \vdash v.m_\delta(v_1, \ldots, v_i, e_{i+1}, \ldots, e_n) \; R \; \bar{\Psi} \vdash v.m_\delta(v_1, \ldots, v_i, e'_{i+1}, \ldots, e_n)}$$

$$\frac{\begin{array}{c}v = C(\ldots) \\ C.m_\delta(\tau_1\ x_1, \ldots, \tau_n\ x_n) \mapsto e \in \bar{\Psi} \qquad (\bar{\Psi} \vdash v \triangleleft \delta) \qquad (\forall i.\ \bar{\Psi} \vdash v_i \triangleleft \tau_i)\end{array}}{\bar{\Psi} \vdash v.m_\delta(v_1, \ldots, v_n) \; R \; e[\texttt{this} \mapsto v, x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]}$$

$$\frac{\bar{\Psi} \vdash e \; R \; e'}{\bar{\Psi} \vdash \texttt{cast}\ e\ \texttt{to}\ \tau \; R \; \texttt{cast}\ e'\ \texttt{to}\ \tau} \qquad \frac{v = C(\ldots) \qquad \bar{\Psi} \vdash C \blacktriangleleft \tau \qquad (\bar{\Psi} \vdash v \triangleleft \top)}{\bar{\Psi} \vdash \texttt{cast}\ v\ \texttt{to}\ \tau \; R \; v}$$

Figure B.2: Reduction Rules without Evaluation Contexts, where R is either optimistic $\rightarrowtriangle$ (ignoring parenthesized assumptions) or pessimistic $\twoheadrightarrow$ (asserting parenthesized assumptions) reduction

This formalization is also equivalent to the one in Figure 5.6, so we abuse notation again and denote both judgements in the same manner.

B.2.1 *Progress*

Lemma B.1. *For every environment* Ψ *where* $\vdash \Psi$ *holds,*

$$\forall v, \tau. \quad \Psi \vdash v \triangleleft \tau \quad \Longrightarrow \quad \Psi \vdash v \blacktriangleleft \tau$$

Proof. By induction on the proof of $\Psi \vdash v \triangleleft \tau$, applying the fact that $\Psi \vdash C \triangleleft \tau$ implies $\Psi \vdash C \blacktriangleleft \tau$ for any Ψ, C, and τ. □

Lemma B.2. *For every environment* Ψ *and implementation* $\bar{\Psi}$ *where* $\vdash \Psi$ *and* $\Psi \vdash_S \bar{\Psi}$ *hold,*

$$\forall e, \tau. \quad \Psi \vdash e \; S \; \tau \quad \Longrightarrow \quad \begin{array}{c} \exists v. \quad e = v \quad \wedge \quad \Psi \vdash v \blacktriangleleft \tau \\ \textit{or} \\ \exists \varepsilon. \quad \Psi \vdash e \; \texttt{erroneous} \; \varepsilon \\ \textit{or} \\ \exists e'. \quad \bar{\Psi} \vdash e \; R \; e' \end{array}$$

where S *is either optimistic* $\triangleleft$ *or pessimistic* $\blacktriangleleft$ *subtyping, and* R *is either optimistic* $\rightarrowtriangle$ *or pessimistic* $\rightarrow$ *reduction.*

Proof. By induction on the proof of $\Psi \vdash e \; S \; \tau$, applying Lemma B.1 when necessary. Note that the first two cases combined are simply $\Psi \vdash e \; \texttt{terminal} \; \tau$. □

Lemma B.3. *For every environment* Ψ *and implementation* $\bar{\Psi}$ *where* $\vdash \Psi$ *and* $\Psi \vdash_S \bar{\Psi}$ *hold,*

$$\forall \Psi, e, \tau. \quad \Psi \vdash e \ \texttt{terminal} \ \tau \quad \Longrightarrow \quad \nexists e'. \quad \bar{\Psi} \vdash e \ R \ e'$$

where S *is either optimistic* $\triangleleft$ *or pessimistic* $\blacktriangleleft$ *subtyping, and* R *is either optimistic* $\multimap$ *or pessimistic* $\rightarrow$ *reduction.*

Proof. By induction on the two cases of $\Psi \vdash e \ \texttt{terminal} \ \tau$. □

Theorem 5.5.1 (Progress)**.** *For every environment* Ψ *and implementation* $\bar{\Psi}$ *where* $\vdash \Psi$ *and* $\Psi \vdash_S \bar{\Psi}$ *hold,*

$$\forall e, \tau. \quad \Psi \vdash e \ S \ \tau \quad \Longrightarrow \quad \Psi \vdash e \ \texttt{terminal} \ \tau \quad \mathit{xor} \quad \exists e'. \quad \bar{\Psi} \vdash e \ R \ e'$$

where S *is either optimistic* $\triangleleft$ *or pessimistic* $\blacktriangleleft$ *subtyping, and* R *is either optimistic* $\multimap$ *or pessimistic* $\rightarrow$ *reduction.*

Proof. Corollary of Lemmas B.2 and B.3. □

B.2.2 *Pessimistic-Type Preservation*

Lemma B.4. *For every environment* Ψ *where* $\vdash \Psi$ *holds,*

$$\forall \tau, \tau', \tau''. \quad \Psi \vdash \tau \blacktriangleleft \tau' \quad \wedge \quad \Psi \vdash \tau' \blacktriangleleft \tau'' \quad \Longrightarrow \quad \Psi \vdash \tau \blacktriangleleft \tau''$$

Proof. By consideration of the cases of both $\Psi \vdash \tau \blacktriangleleft \tau'$ and $\Psi \vdash \tau' \blacktriangleleft \tau''$. □

Lemma B.5. *For every environment* Ψ *where* $\vdash \Psi$ *holds,*

$$\forall \Gamma, e, \tau, \tau'. \quad \Psi \mid \Gamma \vdash e \blacktriangleleft \tau \quad \wedge \quad \Psi \vdash \tau \blacktriangleleft \tau' \quad \Longrightarrow \quad \Psi \mid \Gamma \vdash e \blacktriangleleft \tau'$$

Proof. By induction on the proof of $\Psi \mid \Gamma \vdash e \blacktriangleleft \tau$, regularly applying Lemma B.4. □

Lemma B.6. *For every environment Ψ where $\vdash \Psi$ holds, and for all Γ, τ_x, x, e, τ, and e_x,*

$$\Psi \mid \Gamma, \tau_x\ x \vdash e \blacktriangleleft \tau \quad \wedge \quad \Psi \mid \Gamma \vdash e_x \blacktriangleleft \tau_x \quad \Longrightarrow \quad \Psi \mid \Gamma \vdash e[x \mapsto e_x] \blacktriangleleft \tau$$

Proof. By induction on the proof of $\Psi \mid \Gamma, \tau_x\ x \vdash e \blacktriangleleft \tau$, applying Lemma B.5 in the case where e is x. □

Theorem 5.5.2 (Pessimistic-Type Preservation)**.** *For every environment Ψ and implementation $\bar{\Psi}$ where $\vdash \Psi$ and $\Psi \vdash_{\blacktriangleleft} \bar{\Psi}$ hold,*

$$\forall \tau, e, e'.\quad \Psi \vdash \tau \quad \wedge \quad \Psi \vdash e \blacktriangleleft \tau \quad \wedge \quad \bar{\Psi} \vdash e\ R\ e' \quad \Longrightarrow \quad \Psi \vdash e' \blacktriangleleft \tau$$

where R is either optimistic $\twoheadrightarrow$ or pessimistic $\rightarrow$ reduction.

Proof. By induction on the proof of $\bar{\Psi} \vdash e\ R\ e'$, applying Lemma B.6 in cases with variable substitutions. □

B.2.3 *Pessimistic Identification*

Lemma B.7. *For every environment Ψ and implementation $\bar{\Psi}$ where $\vdash \Psi$ and $\Psi \vdash_S \bar{\Psi}$ hold,*

$$\forall e, \tau.\quad \Psi \vdash e\ S\ \tau \quad \Longrightarrow \quad \forall e'.\quad \bar{\Psi} \vdash e \twoheadrightarrow e' \quad \Longleftrightarrow \quad \bar{\Psi} \vdash e \rightarrow e'$$

where S is either optimistic $\triangleleft$ or pessimistic $\blacktriangleleft$ subtyping.

Proof. Pessimistic reduction always implies optimistic reduction trivially from the respective definitions. When the term being reduced is optimistically typed, optimistic reduction implies pessimistic reduction because the only additional requirements of pessimistic reduction are that the components of the expression that are relevant to the reduction are appropriately typed, which trivially holds when the entire expression is optimistically typed. When the term being reduced is pessimistically typed, it is trivially also optimistically typed, so the same reasoning applies. □

Theorem 5.5.3 (Pessimistic Identification)**.** *For every environment* Ψ *and implementation* $\bar{\Psi}$ *where* $\vdash \Psi$ *and* $\Psi \vdash_{\blacktriangleleft} \bar{\Psi}$ *hold,*

$$\forall e, \tau. \quad \Psi \vdash e \blacktriangleleft \tau \quad \Longrightarrow \quad \forall v. \quad \bar{\Psi} \vdash e \rightarrow\!\!\!\!\triangleright^{\infty} v : \tau \quad \Longleftrightarrow \quad \bar{\Psi} \vdash e \rightarrow\!\!\!\!\blacktriangleright^{\infty} v : \tau$$

Proof. Corollary of Lemma B.7 and Theorem 5.5.2. □

B.3 PROOF OF SEMANTIC PRESERVATION

The full rules for program refinement are presented in Figure B.3.

B.3.1 *Translation Irrelevance*

Theorem 5.6.1 (Translation Irrelevance)**.** *For every* Ψ, $\bar{\Psi}_1$, $\bar{\Psi}_2$, e, $\tilde{e}_1$, $\tilde{e}_2$, *and* τ,

$$\begin{pmatrix} \vdash \Psi \mid e \rightsquigarrow \bar{\Psi}_1 \mid \tilde{e}_1 : \tau \\ \vdash \Psi \mid e \rightsquigarrow \bar{\Psi}_2 \mid \tilde{e}_2 : \tau \end{pmatrix} \Longrightarrow \quad \forall v.\ \bar{\Psi}_1 \vdash \tilde{e}_1\ R^{\infty}\ v : \tau \Longleftrightarrow \bar{\Psi}_2 \vdash \tilde{e}_2\ R^{\infty}\ v : \tau$$

where R *is either optimistic* $\rightarrow\!\!\!\!\triangleright$ *or pessimistic* $\rightarrow\!\!\!\!\blacktriangleright$ *reduction.*

Expression Refinement $\Psi \vdash e \preceq e : \tau$

$$\frac{}{\Psi \vdash x \preceq x : \tau} \qquad \frac{\Psi \vdash e_x \preceq \tilde{e}_x : \tau_x \qquad \Psi \vdash e \preceq \tilde{e} : \tau}{\Psi \vdash \textbf{let}\ \tau_x\ x \coloneqq e_x\ \textbf{in}\ e \preceq \textbf{let}\ \tau_x\ x \coloneqq \tilde{e}_x\ \textbf{in}\ \tilde{e} : \tau}$$

$$\frac{\textbf{class}\ C(\tau_1, ..., \tau_n) \in \Psi \qquad \forall i.\ \Psi \vdash e_i \preceq \tilde{e}_i : \tau_i}{\Psi \vdash C(e_1, \ldots, e_n) \preceq C(\tilde{e}_1, \ldots, \tilde{e}_n) : \tau} \qquad \frac{\Psi \vdash e \preceq \tilde{e} : \delta}{\Psi \vdash e.f_\delta \preceq \tilde{e}.f_\delta : \tau}$$

$$\frac{\Psi \vdash \delta.m(\tau_1, ..., \tau_n) : \tau \qquad \Psi \vdash e \preceq \tilde{e} : \delta \qquad \forall i.\ \Psi \vdash e_i \preceq \tilde{e}_i : \tau_i}{\Psi \vdash e.m_\delta(e_1, \ldots, e_n) \preceq \tilde{e}.m_\delta(\tilde{e}_1, \ldots, \tilde{e}_n) : \tau'}$$

$$\frac{\Psi \vdash e \preceq \tilde{e} : \top}{\Psi \vdash \textbf{cast}\ e\ \textbf{to}\ \tau \preceq \textbf{cast}\ \tilde{e}\ \textbf{to}\ \tau : \tau'}$$

$$\frac{\Psi \vdash e \preceq \tilde{e} : \tau}{\Psi \vdash e \preceq \textbf{cast}\ \tilde{e}\ \textbf{to}\ \tau : \tau} \qquad \frac{\Psi \vdash e \preceq \tilde{e} : \tau \qquad e \neq \tilde{e}}{\Psi \vdash e \prec \tilde{e} : \tau}$$

Program Refinement $\Psi \vdash \Psi \preceq \Psi$

$$\frac{}{\Psi \vdash \cdot \preceq \cdot} \qquad \frac{\Psi \vdash \bar{\Psi} \preceq \bar{\Psi}'}{\Psi \vdash \bar{\Psi}, i \preceq \bar{\Psi}', i} \qquad \frac{\Psi \vdash \bar{\Psi} \preceq \bar{\Psi}' \qquad \Psi \vdash \bar{c} \preceq \bar{c}'}{\Psi \vdash \bar{\Psi}, \bar{c} \preceq \bar{\Psi}', \bar{c}'}$$

Class Refinement $\Psi \vdash \bar{c} \preceq \bar{c}$

$$\frac{\begin{array}{c}\bar{c} = \textbf{class}\ C(\tau_1\ f_1, \ldots)\ \textbf{implements}\ C_1, ...\ \{\bar{d}_1; \ldots\} \\ \bar{c}' = \textbf{class}\ C(\tau_1\ f_1, \ldots)\ \textbf{implements}\ C_1, ...\ \{\bar{d}'_1; \ldots\} \qquad \forall i.\ \Psi \vdash \bar{d}_i \preceq \bar{d}'_i\end{array}}{\Psi \vdash \bar{c} \preceq \bar{c}'}$$

Method Refinement $\Psi \vdash \bar{d} \preceq \bar{d}$

$$\frac{\Psi \vdash e \preceq \tilde{e} : \tau}{\Psi \vdash \tau\ m_\delta(\Gamma) \mapsto e \preceq \tau\ m_\delta(\Gamma) \mapsto \tilde{e}}$$

Figure B.3: Program Refinement

Proof. Given two refinements $\tilde{e}$ and $\tilde{e}'$ of an expression e, one can construct an expression $\tilde{e}''$ that is a refinement of both $\tilde{e}$ and $\tilde{e}'$. Likewise, given two implementations $\bar{\Psi}$ and $\bar{\Psi}'$ of an environment Ψ, one can construct an implementation $\bar{\Psi}''$ that is a refinement of both $\bar{\Psi}$ and $\bar{\Psi}'$. One can generalize the lemmas of this appendix to also relate the semantics between refinements of implementations. A corollary of those generalizations, specifically the generalizations of Lemmas B.12 and B.17, is that a pessimistically typed program will have the exact same semantics as any of its refinements due to Theorem 5.5.2. Thus $\tilde{e}$ must have the same semantics as its refinement $\tilde{e}''$, which must have the same semantics as $\tilde{e}'$, implying $\tilde{e}$ and $\tilde{e}'$ have the same semantics. □

B.3.2 *Translation Existence*

Theorem 5.6.2 (Translation Existence)**.** *For every environment* Ψ*, expression* e*, and type* τ*,*

$$\vdash \Psi \quad \wedge \quad \Psi \vdash \tau \ \wedge \quad \Psi \vdash e \triangleleft \tau \quad \Longrightarrow \quad \exists \bar{\Psi}, \tilde{e}. \quad \vdash \Psi \mid e \rightsquigarrow \bar{\Psi} \mid \tilde{e} : \tau$$

Proof. An algorithm for developing well-formed translations is presented in Figure B.4. Note that this algorithm is naïve in that it nearly always inserts casts regardless of whether they may actually be necessary. The only major exception is the case of class constructors, in which the fact that the expression optimistically has the expected return type implies C is a pessimistic subtype of the expected return type. Regardless, even though there are more complex algorithms that would produce more efficient translations, this one is still correct. □

Expression Translation $\Psi \mid \Gamma \vdash e \rightsquigarrow e : \tau$

$$\frac{}{\Psi \mid \Gamma \vdash x \rightsquigarrow \mathbf{cast}\ x\ \mathbf{to}\ \tau : \tau}$$

$$\frac{\Psi \mid \Gamma \vdash e_x \rightsquigarrow \tilde{e}_x : \tau \qquad \Psi \mid \Gamma, \tau\ x \vdash e \rightsquigarrow \tilde{e} : \tau'}{\Psi \mid \Gamma \vdash \mathbf{let}\ \tau\ x := e_x\ \mathbf{in}\ e \rightsquigarrow \mathbf{let}\ \tau\ x := \tilde{e}_x\ \mathbf{in}\ \tilde{e} : \tau'}$$

$$\frac{\mathbf{class}\ C(\tau_1, ..., \tau_n) \in \Psi \qquad \forall i.\ \Psi \mid \Gamma \vdash e_i \rightsquigarrow \tilde{e}_i : \tau_i}{\Psi \mid \Gamma \vdash C(e_1, \ldots, e_n) \rightsquigarrow C(\tilde{e}_1, \ldots, \tilde{e}_n) : \tau}$$

$$\frac{\Psi \mid \Gamma \vdash e \rightsquigarrow \tilde{e} : \delta}{\Psi \mid \Gamma \vdash e.f_\delta \rightsquigarrow \mathbf{cast}\ \tilde{e}.f_\delta\ \mathbf{to}\ \tau : \tau}$$

$$\frac{\Psi \vdash \delta.m(\tau_1, \ldots, \tau_n) : \tau \qquad \Psi \mid \Gamma \vdash e \rightsquigarrow \tilde{e} : \delta \qquad \forall i.\ \Psi \mid \Gamma \vdash e_i \rightsquigarrow \tilde{e}_i : \tau_i}{\Psi \mid \Gamma \vdash e.m_\delta(e_1, \ldots, e_n) \rightsquigarrow \mathbf{cast}\ \tilde{e}.m_\delta(\tilde{e}_1, \ldots, \tilde{e}_n)\ \mathbf{to}\ \tau' : \tau'}$$

$$\frac{\Psi \mid \Gamma \vdash e \rightsquigarrow \tilde{e} : \top}{\Psi \mid \Gamma \vdash \mathbf{cast}\ e\ \mathbf{to}\ \tau \rightsquigarrow \mathbf{cast}\ \mathbf{cast}\ \tilde{e}\ \mathbf{to}\ \tau\ \mathbf{to}\ \tau' : \tau'}$$

Program Translation $\vdash \Psi \rightsquigarrow \Psi \qquad \Psi \vdash \Psi \rightsquigarrow \Psi$

$$\frac{\Psi \vdash \Psi \rightsquigarrow \bar{\Psi}}{\vdash \Psi \rightsquigarrow \bar{\Psi}} \qquad \frac{}{\Psi \vdash \cdot \rightsquigarrow \cdot} \qquad \frac{\Psi \vdash \Psi' \rightsquigarrow \bar{\Psi}}{\Psi \vdash \Psi', i \rightsquigarrow \bar{\Psi}, i} \qquad \frac{\Psi \vdash \Psi' \rightsquigarrow \bar{\Psi} \qquad \Psi \vdash c \rightsquigarrow \bar{c}}{\Psi \vdash \Psi', c \rightsquigarrow \bar{\Psi}, \bar{c}}$$

Class Translation $\Psi \vdash c \rightsquigarrow \bar{c}$

$$\frac{\begin{array}{c} c = \mathbf{class}\ C(\tau_1\ f_1, \ldots)\ \mathbf{implements}\ C_1, \ldots\ \{d_1; \ldots\} \\ \bar{c} = \mathbf{class}\ C(\tau_1\ f_1, \ldots)\ \mathbf{implements}\ C_1, \ldots\ \{\bigcup_{\delta \in \{C, \mathbf{dyn}, C_1, \ldots\}} \bar{d}_1^{\delta}; \ldots\} \\ \forall i.\ \Psi \mid C \vdash d_i \rightsquigarrow \bar{d}_i^{C} : d_i \\ \forall i.\ \Psi \mid C \vdash d_i \rightsquigarrow \bar{d}_i^{\mathbf{dyn}} : \mathbf{dyn} \qquad \forall i.\ \mathbf{interface}\ C_i\ \{s_1^i; \ldots\} \in \Psi \\ \forall i.\ \forall j.\ \forall k.\ \Psi \vdash d_k : s_j^i \implies \Psi \mid C \vdash d_k \rightsquigarrow \bar{d}_k^{C_i} :_{C_i} s_j^i \end{array}}{\Psi \vdash c \rightsquigarrow \bar{c}}$$

Figure B.4: Program Translation

Method Translation

$$\Psi \mid C \vdash d \rightsquigarrow \bar{d} : d$$
$$\Psi \mid C \vdash d \rightsquigarrow \bar{d} : \mathbf{dyn}$$
$$\Psi \mid C \vdash d \rightsquigarrow_{\delta} \bar{d} : s$$

$$\frac{\Psi \mid C \vdash d \rightsquigarrow_{C} \bar{d} : s}{\Psi \mid C \vdash d \rightsquigarrow \bar{d} : s \mapsto e}$$

$$\frac{\Psi \mid C \vdash \tau\ m(\tau_1\ x_1, \ldots, \tau_n\ x_n) \mapsto e \rightsquigarrow_{\mathbf{dyn}} \bar{d} : \mathbf{dyn}\ m(\mathbf{dyn}\ x_1, \ldots, \mathbf{dyn}\ x_n)}{\Psi \mid C \vdash \tau\ m(\tau_1\ x_1, \ldots, \tau_n\ x_n) \mapsto e \rightsquigarrow \bar{d} : \mathbf{dyn}}$$

$$\frac{\begin{array}{c} d = \tau\ m(\tau_1\ x_1, \ldots, \tau_n\ x_n) \mapsto e \\ \bar{d} = \tau'\ m_{\delta}(\tau'_1\ x_1, \ldots, \tau'_n\ x_n) \mapsto \mathbf{let}\ \tau'\ x := \tilde{e}\ \mathbf{in}\ x \\ e' = \mathbf{let}\ \tau_1\ x_1 := x_1\ \mathbf{in}\ \ldots \mathbf{let}\ \tau_n\ x_n := x_n\ \mathbf{in}\ \mathbf{let}\ \tau\ x := e\ \mathbf{in}\ x \\ \Psi \mid C\ \mathbf{this}, \tau'_1\ x_1, \ldots, \tau'_n\ x_n \vdash e' \rightsquigarrow \tilde{e} : \tau' \end{array}}{\Psi \mid C \vdash d \rightsquigarrow_{\delta} \bar{d} : \tau'\ m(\tau'_1\ x_1, \ldots, \tau'_n\ x_n)}$$

Figure B.4 (contd.): Program Translation

B.3.3 *Pessimistic-Valuation Preservation*

Lemma B.8. *For every environment* Ψ *and implementation* $\bar{\Psi}$ *where* $\vdash \Psi$ *and* $\Psi \vdash_{\blacktriangleleft} \bar{\Psi}$ *hold,*

$$\forall v, \tilde{e}, \tau. \quad \Psi \vdash v \blacktriangleleft \tau \quad \wedge \quad \Psi \vdash v \preceq \tilde{e} : \tau \quad \Longrightarrow \quad \bar{\Psi} \vdash \tilde{e} \twoheadrightarrow^{*} v$$

Proof. Because v is an optimistically typed value, Lemma B.1 tells us that v is also pessimistically typed. Using that fact, the remainder is proven by induction on the proof of $\Psi \vdash v \preceq \tilde{e} : \tau$. □

Lemma B.9. *For every environment* Ψ *and implementation* $\bar{\Psi}$ *where* $\vdash \Psi$ *and* $\Psi \vdash_{\blacktriangleleft} \bar{\Psi}$ *hold,*

$$\forall \varepsilon, e, \tilde{e}, \tau. \quad \begin{pmatrix} \Psi \vdash e \ \textbf{\textit{erroneous}} \ \varepsilon \\ \Psi \vdash e \preceq \tilde{e} : \tau \end{pmatrix} \quad \Longrightarrow \quad \exists \tilde{e}'. \quad \begin{pmatrix} \bar{\Psi} \vdash \tilde{e}' \ \textbf{\textit{erroneous}} \ \varepsilon \\ \bar{\Psi} \vdash \tilde{e} \twoheadrightarrow^{*} \tilde{e}' \end{pmatrix}$$

Proof. By induction on the proof of $\Psi \vdash e \preceq \tilde{e} : \tau$, applying Lemma B.8 to reduce values. □

Lemma B.10. *For every environment* Ψ *where* $\vdash \Psi$ *holds,*

$$\forall e, \tilde{e}, \tau, x, v. \quad \Psi \vdash e \preceq \tilde{e} : \tau \quad \Longrightarrow \quad \Psi \vdash e[x \mapsto v] \preceq \tilde{e}[x \mapsto v] : \tau$$

Proof. By induction on the proof of $\Psi \vdash e \preceq \tilde{e} : \tau$. □

Lemma B.11. *For every environment* Ψ *and implementation* $\bar{\Psi}$ *where* $\vdash \Psi$ *and* $\Psi \vdash_{\blacktriangleleft} \bar{\Psi}$ *hold,*

$$\forall \tau, e, \tilde{e}, e'. \quad \begin{pmatrix} \Psi \vdash e \preceq \tilde{e} : \tau \\ \Psi \vdash e \twoheadrightarrow e' \end{pmatrix} \quad \Longrightarrow \quad \exists \tilde{e}'. \quad \begin{pmatrix} \Psi \vdash e' \preceq \tilde{e}' : \tau \\ \bar{\Psi} \vdash \tilde{e} \twoheadrightarrow^{+} \tilde{e}' \end{pmatrix}$$

Proof. By induction on the proof of $\Psi \vdash e \preceq \tilde{e} : \tau$, applying Lemma B.8 and the assertions of pessimistic reduction to reduce values, and applying Lemma B.10 in cases with variable substitutions.

Note that the case of method-invocation refinement relies on a particular formal detail of implementations. Suppose e is of the form $v.m_{\delta}(v_1, \ldots, v_n)$, where v is an instance of class C, and e pessimistically steps. The fact that this pessimistically steps informs us that each value has its expected type, so we can apply Lemma B.8 to show that $\tilde{e}$ pessimistically reduces to $v.m_{\delta}(v_1, \ldots, v_n)$. It may seem trivial that this then steps to the same

expression that e steps to, but recall that we are reducing e in implementation Ψ but $\tilde{e}$ in implementation $\bar{\Psi}$. Our definition of implementation validation in Figure 5.7 ensures that e steps to $\texttt{let}\ \tau_m^\delta\ x := e_m^C\ \texttt{in}\ x$, where e_m^C is essentially C's definition of m (with arguments substituted), and τ_m^δ is δ's return type for m. Furthermore, our definition of implementation validation ensures that $\tilde{e}$ steps to $\texttt{let}\ \tau_m^\delta\ x := \tilde{e}_m^\delta\ \texttt{in}\ x$, where $\tilde{e}_m^\delta$ is a refinement of e_m^C under type τ_m^δ. Consequently, our definition of implementation validation makes it trivial to show that $\texttt{let}\ \tau_m^\delta\ x := \tilde{e}_m^\delta\ \texttt{in}\ x$ is a refinement of $\texttt{let}\ \tau_m^\delta\ x := e_m^C\ \texttt{in}\ x$ under τ, as required for our goal. However, had our definition of implementation validation elided the use of $\texttt{let}\ \tau_m^\delta\ x := \bullet\ \texttt{in}\ x$ and instead just used $\bullet$, we would have a problem since $\tilde{e}_m^\delta$ is a refinement of e_m^C under τ_m^δ but *not* under the type τ required for our goal. Thus this formal detail is critical to this lemma, but overall it seems to be simply an artifact of our choice of strategy for formalization rather than anything with deep significance. □

Lemma B.12. *For every Ψ, $\bar{\Psi}$, e, $\tilde{e}$, and τ where $\vdash \Psi \mid e \rightsquigarrow \bar{\Psi} \mid \tilde{e} : \tau$ holds,*

$$\forall v.\quad \Psi \vdash e \twoheadrightarrow^\infty v : \tau \quad\Longrightarrow\quad \bar{\Psi} \vdash \tilde{e}\ R^\infty\ v : \tau$$

where R is either optimistic $\twoheadrightarrow$ or pessimistic $\rightarrow$ reduction.

Proof. Since $\tilde{e}$ is pessimistically typed, by Theorem 5.5.3 we only need to prove this for the case where R is pessimistic reduction. This case is a corollary of Lemma B.11, applying Lemma B.8 for the case where v is a value, and Lemma B.9 for the case where v is erroneous, and noting that Lemma B.11 guarantees the translation makes at least one step for every step of the original program for the case where v is non-termination. □

Theorem 5.6.3 (Pessimistic-Valuation Preservation)**.** *For every environment* Ψ*, expression* e*, and type* τ *where* $\vdash \Psi$ *and* $\Psi \vdash \tau$ *and* $\Psi \vdash e \triangleleft \tau$ *hold,*

$$\forall v. \quad \Psi \vdash e \rightarrow^{\infty} v : \tau \quad \Longrightarrow \quad \Psi \vdash e \rightsquigarrow^{\infty} v : \tau$$

Proof. Corollary of Theorem 5.6.2 and Lemma B.12. □

B.3.4 *Optimistic-Valuation Reflection*

The use of negation in our definition of lapses often complicate proofs unnecessarily. In Figure B.5, we present a formalization of lapses without negation except for optimistic subtyping. This formalization is equivalent to the one in Figure 5.5, so we abuse notation and denote both judgements in the same manner.

Lemma B.13. *For every environment* Ψ *where* $\vdash \Psi$ *holds,*

$$\forall e, v, \tau. \quad \Psi \vdash e \preceq v : \tau \quad \Longrightarrow \quad e = v$$

Proof. By induction on the proof of $\Psi \vdash e \preceq v : \tau$. □

Lemma B.14. *For every environment* Ψ *and implementation* $\bar{\Psi}$ *where* $\vdash \Psi$ *and* $\Psi \vdash_{\blacktriangleleft} \bar{\Psi}$ *hold, and for all* e*,* $\tilde{e}$*,* τ*, and* $\tilde{e}'$*,*

$$\Psi \vdash e \preceq \tilde{e} : \tau \ \wedge \ \bar{\Psi} \vdash \tilde{e} \ R \ \tilde{e}' \quad \Longrightarrow \quad \begin{array}{c} \Psi \vdash e \preceq \tilde{e}' : \tau \ \wedge \ \Psi \vdash \tilde{e}' \prec \tilde{e} : \tau \\ \textit{or} \\ \exists e'. \ \Psi \ e \ R \ e' \quad \wedge \quad \Psi \vdash e' \preceq \tilde{e}' : \tau \end{array}$$

where R *is either optimistic* $\twoheadrightarrow$ *or pessimistic* $\rightarrow$ *reduction.*

Lapses without Reduction $\Psi \vdash e \ \mathbf{lapse} \ \tau$

$$\frac{\Psi \vdash e \ \mathbf{lapse} \ \tau}{\Psi \vdash \mathbf{let} \ \tau \ x := e \ \mathbf{in} \ e' \ \mathbf{lapse} \ \tau'} \qquad \frac{\nexists \tau_1, \ldots, \tau_n. \ \mathbf{class} \ C(\tau_1, \ldots, \tau_n) \in \Psi}{\Psi \vdash C(e_1, \ldots, e_n) \ \mathbf{lapse} \ \tau}$$

$$\frac{\mathbf{class} \ C(\tau_1, \ldots, \tau_n) \in \Psi \quad \forall j. \ \Psi \vdash v_j \triangleleft \tau_j \qquad \Psi \vdash e_{i+1} \ \mathbf{lapse} \ \tau_{i+1}}{\Psi \vdash C(v_1, \ldots, v_i, e_{i+1}, \ldots, e_n) \ \mathbf{lapse} \ \tau'} \qquad \frac{\mathbf{class} \ C(\tau_1, \ldots, \tau_n) \in \Psi \quad \forall i. \ \Psi \vdash v_i \triangleleft \tau_i \qquad \neg \ \Psi \vdash C \triangleleft \tau}{\Psi \vdash C(v_1, \ldots, v_n) \ \mathbf{lapse} \ \tau}$$

$$\frac{\Psi \vdash e \ \mathbf{lapse} \ \delta}{\Psi \vdash e.f_\delta \ \mathbf{lapse} \ \tau} \qquad \frac{\Psi \vdash e \ \mathbf{lapse} \ \delta}{\Psi \vdash e.m_\delta(e_1, \ldots, e_n) \ \mathbf{lapse} \ \tau}$$

$$\frac{\Psi \vdash v \triangleleft \delta \qquad \nexists \tau_1, \ldots, \tau_n, \tau. \ \Psi \vdash \delta.m(\tau_1, \ldots, \tau_n) : \tau}{\Psi \vdash v.m_\delta(e_1, \ldots, e_n) \ \mathbf{lapse} \ \tau'}$$

$$\frac{\Psi \vdash \delta.m(\tau_1, \ldots, \tau_n) : \tau \quad \Psi \vdash v \triangleleft \delta \qquad \forall j. \ \Psi \vdash v_j \triangleleft \tau_j \qquad \Psi \vdash e_{i+1} \ \mathbf{lapse} \ \tau_{i+1}}{\Psi \vdash v.m_\delta(v_1, \ldots, v_i, e_{i+1}, \ldots, e_n) \ \mathbf{lapse} \ \tau'}$$

$$\frac{\Psi \vdash e \ \mathbf{lapse} \ \top}{\Psi \vdash \mathbf{cast} \ e \ \mathbf{to} \ \tau \ \mathbf{lapse} \ \tau'}$$

Figure B.5: Lapses without Reduction

Proof. Proof by induction on the proof of $\Psi \vdash e \preceq \tilde{e} : \tau$, applying Lemma B.13 to get values, and applying Lemma B.10 in cases with variable substitutions. The case for method invocation relies on the same detail of implementations as Lemma B.11, and the proof is nearly identical. Note that the fact that this lemma holds for pessimistic reduction, not just optimistic reduction, is the stronger property of our system that enables us to achieve immediacy. □

Lemma B.15. *For every environment* Ψ *and implementation* $\bar{\Psi}$ *where* $\vdash \Psi$ *and* $\Psi \vdash_{\blacktriangleleft} \bar{\Psi}$ *hold, and for each* ε, e, $\tilde{e}$, *and* τ,

$$\left(\begin{array}{c} \bar{\Psi} \vdash \tilde{e}\ \textbf{\texttt{erroneous}}\ \varepsilon \\ \\ \Psi \vdash e \preceq \tilde{e} : \tau \end{array}\right) \implies \begin{array}{c} \Psi \vdash e\ \textbf{\texttt{erroneous}}\ \varepsilon \\ \\ \textit{or} \\ \\ \Psi \vdash \varepsilon\ \textbf{\texttt{bad-cast}} \ \wedge\ \Psi \vdash e\ \textbf{\texttt{lapse}}\ \tau \end{array}$$

Proof. By induction on the proof of $\Psi \vdash e \preceq \tilde{e} : \tau$, applying Lemma B.13 to get values. □

Lemma B.16. *For every environment* Ψ *and type* τ *where* $\vdash \Psi$ *and* $\Psi \vdash \tau$ *hold, the binary relation* $\Psi \vdash e \prec e' : \tau$ *is well-founded.*

Proof. Given any e and e' such that $\Psi \vdash e \preceq e' : \tau$ holds, we can show by induction on the proof of $\Psi \vdash e \preceq e' : \tau$ that either the syntactic height of e is strictly less than the syntactic height of e' or the expressions e and e' are syntactically identical. Consequently, for any e and e' such that $\Psi \vdash e \prec e' : \tau$ holds, the expressions e and e' are by definition distinct, so the syntactic height of e must be strictly less than the syntactic height of e', ensuring well-foundedness. □

Lemma B.17. *For every* Ψ, $\bar{\Psi}$, e, $\tilde{e}$, *and* τ *where* $\vdash \Psi \mid e \rightsquigarrow \bar{\Psi} \mid \tilde{e} : \tau$ *holds,*

$$\forall \nu.\quad \Psi \vdash \tilde{e}\ R^{\infty}\ \nu : \tau \quad \implies \quad \begin{array}{c} \Psi \vdash e\ R^{\infty}\ \nu : \tau \\ \\ \textit{xor} \\ \\ \Psi \vdash \nu\ \textbf{\texttt{bad-cast}} \quad \wedge \quad \Psi \vdash e\ R^{*}\ \textbf{\texttt{lapse}}\ \tau \end{array}$$

where R *is either optimistic* $\twoheadrightarrow$ *or pessimistic* $\rightarrow$ *reduction.*

Proof. Proof by consideration of the cases of $\Psi \vdash \tilde{e}\ R^{\infty}\ \nu : \tau$, applying Lemma B.14 in all cases. In the case of a value, one furthermore applies

Lemma B.13. In the case of an error, one furthermore applies Lemma B.15. In the case of non-termination, one furthermore applies Lemma B.16. □

Theorem 5.6.4 (Optimistic-Valuation Reflection)**.** *For every environment* Ψ*, expression* e*, and type* τ *where* $\vdash \Psi$ *and* $\Psi \vdash \tau$ *and* $\Psi \vdash e \triangleleft \tau$ *hold,*

$$\forall v.\quad \Psi \vdash e \rightsquigarrow^{\infty} v : \tau \quad \Longrightarrow \quad \begin{array}{c} \Psi \vdash e \twoheadrightarrow^{\infty} v : \tau \\ \textit{or} \\ \Psi \vdash v\ \texttt{bad-cast} \quad \wedge \quad \Psi \vdash e \twoheadrightarrow^{*} \texttt{lapse}\ \tau \end{array}$$

Proof. Corollary of Lemma B.17. □

B.4 PROOF OF GUARANTEES

B.4.1 *Immediacy*

Theorem 5.7.1 (Immediacy)**.** *For every* Ψ*,* $\bar{\Psi}$*,* e*,* $\tilde{e}$*, and* τ *where* $\vdash \Psi \mid e \rightsquigarrow \bar{\Psi} \mid \tilde{e} : \tau$ *holds,*

$$\forall e'.\ \begin{pmatrix} \Psi \vdash e \twoheadrightarrow^{*} e' \\ \Psi \vdash e'\ \texttt{lapse}\ \tau \end{pmatrix} \Longrightarrow \exists \tilde{e}'.\ \begin{pmatrix} \bar{\Psi} \vdash \tilde{e} \twoheadrightarrow^{*} \tilde{e}' \\ \bar{\Psi} \vdash \tilde{e}'\ \texttt{bad-cast} \end{pmatrix} \wedge\ \Psi \vdash e' \preceq \tilde{e}' : \tau$$

Proof. By Lemma B.11, $\tilde{e}$ must pessimistically (and therefore optimistically) reduce to some $\tilde{e}'_0$ that is a refinement of e'. By Theorem 5.5.2, $\tilde{e}'_0$ is pessimistically typed since $\tilde{e}$ is pessimistically typed. By combining Lemmas B.14 and B.16, because $\tilde{e}'_0$ is a pessimistically typed refinement of a pessimistically irreducible expression e', there must exist a pessimistically irreducible expression $\tilde{e}'$ such that $\bar{\Psi} \vdash \tilde{e}'_0 \twoheadrightarrow^{*} \tilde{e}'$ and $\Psi \vdash e' \preceq \tilde{e}'$ hold. By

Theorem 5.5.2, $\tilde{e}'$ is also pessimistically typed and so, by Theorem 5.5.1, also terminal.

Now first suppose $\tilde{e}'$ were a value of type τ. Then, by Lemma B.13, e' would have to be that same value of type τ. However, by assumption e' is not a terminal of type τ and so cannot be such a value. Thus $\tilde{e}'$ must be erroneous. Since $\tilde{e}'$ is a refinement of e', but e' is itself not erroneous (since it is not terminal), Lemma B.15 informs us that $\tilde{e}'$ must be specifically a bad cast, proving our theorem. □

Corollary B.1. *For every environment Ψ, expression e, and type τ where $\vdash \Psi$, $\Psi \vdash \tau$, and $\Psi \vdash e \triangleleft \tau$ hold,*

$$\Psi \vdash e \twoheadrightarrow^* \mathbf{lapse}\ \tau \quad \Longrightarrow \quad \exists \varepsilon.\ \ \Psi \vdash e \rightsquigarrow^\infty \varepsilon : \tau \ \ \wedge \ \ \Psi \vdash \varepsilon\ \mathbf{bad\text{-}cast}$$

Proof. Corollary of Theorem 5.7.1. □

Lemma B.18. *For every environment Ψ,*

$$\forall e, e'.\ \ \Psi \vdash e \rightarrowtriangle e' \quad \Longrightarrow \quad \begin{array}{c} \Psi \vdash e \twoheadrightarrow e' \\ \textit{or} \\ \forall \tau.\ \ \Psi \vdash e\ \mathbf{lapse}\ \tau \end{array}$$

Proof. By induction on the proof of $\Psi \vdash e \rightarrowtriangle e'$. □

Lemma B.19 (Pessimistic-Valuation Reflection)**.** *For every environment Ψ, expression e, and type τ where $\vdash \Psi$ and $\Psi \vdash \tau$ and $\Psi \vdash e \triangleleft \tau$ hold,*

$$\forall v.\ \ \Psi \vdash e \rightsquigarrow^\infty v : \tau \quad \Longrightarrow \quad \begin{array}{c} \Psi \vdash e \twoheadrightarrow^\infty v : \tau \\ \textit{or} \\ \Psi \vdash v\ \mathbf{bad\text{-}cast} \ \ \wedge \ \ \Psi \vdash e \twoheadrightarrow^* \mathbf{lapse}\ \tau \end{array}$$

Proof. Note that this can be proven as a corollary of Theorem 5.5.3 and Lemma B.17. However, the proof of Lemma B.17 with pessimistic reduction relies on the fact that Lemma B.14 holds with pessimistic reduction. For many gradual type systems, Lemma B.14 only holds for optimistic reduction. So our goal here is to show that pessimistic-valuation reflection can proven from just immediacy and properties we expect to hold of any sound gradual type system.

Given $\Psi \vdash e \rightsquigarrow^{\infty} v : \tau$, Theorem 5.6.4 tells us that either e optimistically results in v or v is a bad cast and e optimistically reduces to a lapse. In more detail, optimistic-valuation reflection tells us that for each sequence $\tilde{e}_0 \longrightarrow_{\triangleright} \ldots$ of single-step optimistic reductions, where $\tilde{e}_0$ is a pessimistically typed refinement of e, there is a corresponding sequence $e_0 \longrightarrow_{\triangleright}^{?} \ldots$ of single-or-no-step optimistic reductions, where e_0 is e and each $\tilde{e}_i$ is a refinement of e_i. Furthermore, if the first sequence terminates with a value, then the second sequence terminates with that same value. And if the first sequence continues forever, then the second sequence continues forever and contains enough single-step reductions to correspond to an infinite digression. And if the first sequence terminates with an error, then either the second sequence terminates with that error, or the error is a bad cast and the second sequence terminates with some lapse.

Now suppose every optimistic reduction in the second sequence is also a pessimistic reduction. Then the optimistic valuation or lapse of e is also a pessimistic result of e, achieving our goal.

The alternative then, is that there is some single-step optimistic reduction in the second sequence that is not also a pessimistic reduction. Let i be such that $e_i \longrightarrow_{\triangleright} e_{i+1}$ is the first such single-step reduction. Since i is the first such reduction, by definition e pessimistically reduces to e_i. By Lemma B.18,

e_i furthermore lapses. And since e_i is optimistically reducible, it cannot be a value. Thus we can apply Theorem 5.7.1. In more detail, immediacy tells us that $\tilde{e}_i$, being a pessimistically typed refinement of a lapse e_i, necessarily optimistically reduces to a bad cast that is itself a refinement of e_i. In particular, immediacy tells us that the first sequence terminates and every single-or-no-step reduction after e_i is actually a no-step reduction. In particular, $e_i \mathrel{-\!\!\triangleright^?} e_{i+1}$ must be a no-step reduction, contradicting our definition of i. Thus, no such i can exist, guaranteeing that every optimistic reduction in the second sequence is also a pessimistic reduction. □

B.4.2 *Gradual Optimism*

Lemma B.20. *For every environment* Ψ *and* Ψ'*,*

$$\Psi \sqsubseteq \Psi' \implies \forall \tau, \tau'.\ \ \Psi \vdash \tau\ S\ \tau' \implies \Psi' \vdash \tau\ S\ \tau'$$

where S *is either optimistic* $\triangleleft$ *or pessimistic* $\blacktriangleleft$ *subtyping.*

Proof. By induction on $\Psi \sqsubseteq \Psi'$, since precision does not affect the inheritance hierarchy. □

Lemma B.21. *For every environment* Ψ*,*

$$\forall \tau_1, \tau_1', \tau_2, \tau_2'.\ \ \tau_1 \sqsubseteq \tau_1' \wedge \tau_2 \sqsubseteq \tau_2' \wedge \Psi \vdash \tau_1 \triangleleft \tau_2 \implies \Psi \vdash \tau_1' \triangleleft \tau_2'$$

Proof. By consideration of the cases of $\tau_1 \sqsubseteq \tau_1'$, $\tau_2 \sqsubseteq \tau_2'$, and $\Psi \vdash \tau_1 \triangleleft \tau_2$. □

Theorem 5.7.2 (Gradual Optimism)**.**

$$\forall \begin{pmatrix} \Psi, \Psi' \\ \Gamma,\ \Gamma' \\ \tau,\ \tau' \\ e,\ e' \end{pmatrix} . \begin{pmatrix} \vdash \Psi \\ \Psi \quad \vdash \Gamma \\ \Psi \quad \vdash \tau \\ \Psi \mid \Gamma \vdash e \triangleleft \tau \end{pmatrix} \wedge \begin{pmatrix} \Psi \sqsubseteq \Psi' \\ \Gamma \sqsubseteq \Gamma' \\ \tau \sqsubseteq \tau' \\ e \sqsubseteq e' \end{pmatrix} \implies \begin{pmatrix} \vdash \Psi' \\ \Psi' \quad \vdash \Gamma' \\ \Psi' \quad \vdash \tau' \\ \Psi' \mid \Gamma' \vdash e' \triangleleft \tau' \end{pmatrix}$$

Proof. Each typing conclusion is proven by induction on the corresponding typing assumption, regularly applying Lemmas B.20 and B.21. □

B.4.3 *Gradual Preservation*

Lemma B.22.

$$\forall v, e'. \quad v \sqsubseteq e' \quad \implies \quad e' = v$$

Proof. By induction on the proof of $v \sqsubseteq e'$. □

Lemma B.23.

$$\forall e, e', x, v. \quad e \sqsubseteq e' \quad \implies \quad e[x \mapsto v] \sqsubseteq e'[x \mapsto v]$$

Proof. By induction on the proof of $e \sqsubseteq e'$. □

Lemma B.24. *For every environment* Ψ*,*

$$\forall \tau_1, \tau_2, \tau_2'. \quad \tau_2 \sqsubseteq \tau_2' \quad \wedge \quad \Psi \vdash \tau_1 \blacktriangleleft \tau_2 \quad \implies \quad \Psi \vdash \tau_1 \blacktriangleleft \tau_2'$$

Proof. By consideration of the cases of $\tau_2 \sqsubseteq \tau_2'$ and $\Psi \vdash \tau_1 \blacktriangleleft \tau_2$. □

Lemma B.25. *For every* Ψ *and* Ψ' *where* $\Psi \sqsubseteq \Psi'$ *holds,*

$$\forall e_1, e_1', e_2.\quad e_1 \sqsubseteq e_1' \;\wedge\; \Psi \vdash e_1\ R\ e_2 \quad\Longrightarrow\quad \exists e_2'.\quad \Psi' \vdash e_1'\ R\ e_2' \;\wedge\; e_2 \sqsubseteq e_2'$$

where R *is either optimistic* $\multimap$ *or pessimistic* $\rightarrow$ *reduction.*

Proof. By induction on the proof of $\Psi \vdash e_1\ R\ e_2$, applying Lemma B.22 to get values, applying Theorem 5.7.2 to ensure those values are still typed in the case of pessimistic reduction, applying Lemma B.23 in cases with variable substitutions, and applying Lemmas B.20 and B.24 in the case of cast reduction. □

Lemma B.26. *For every* Ψ *and* Ψ' *where* $\Psi \sqsubseteq \Psi$ *holds,*

$$\forall \varepsilon, e, e'.\quad \begin{pmatrix} \Psi \vdash e\ \texttt{erroneous}\ \varepsilon \\ e \sqsubseteq e' \end{pmatrix} \quad\Longrightarrow\quad \begin{array}{c} \Psi' \vdash e'\ \texttt{erroneous}\ \varepsilon \\ \textit{or} \\ \Psi \vdash \varepsilon\ \texttt{bad-cast} \end{array}$$

Proof. By induction on the proof of $\Psi \vdash e\ \texttt{erroneous}\ \varepsilon$, applying Theorem 5.7.2 and Lemmas B.22 and B.1 for values. □

Corollary B.2. *For every* Ψ *and* Ψ' *where* $\Psi \sqsubseteq \Psi'$ *holds,*

$$\forall e, e', v, \tau.\quad \Psi \vdash e\ R^\infty\ v : \tau \quad\wedge\quad e \sqsubseteq e' \quad\Longrightarrow\quad \begin{array}{c} \Psi' \vdash e'\ R^\infty\ v : \tau \\ \textit{or} \\ \Psi \vdash v\ \texttt{bad-cast} \end{array}$$

where R *is either optimistic* $\multimap$ *or pessimistic* $\rightarrow$ *reduction.*

Proof. Proof by consideration of the cases of $\Psi \vdash e\ R^\infty\ v : \tau$, applying Lemma B.25 in all cases. In the case of a value, one furthermore applies

Lemma B.22. In the case of an error, one furthermore applies Lemma B.26.

□

Theorem 5.7.3 (Gradual Preservation). *For every* Ψ, Ψ', e, e', τ, *and* τ' *such that* $\vdash \Psi$, $\vdash \Psi'$, $\Psi \vdash \tau$, $\Psi' \vdash \tau'$, $\Psi \vdash e \triangleleft \tau$, *and* $\Psi' \vdash e' \triangleleft \tau'$ *hold,*

$$\forall v. \quad \begin{pmatrix} \Psi \vdash \ e \rightsquigarrow^{\infty} v : \tau \\ \Psi \ \sqsubseteq \ \Psi' \\ \tau \ \sqsubseteq \ \tau' \\ e \ \sqsubseteq \ e' \end{pmatrix} \implies \exists v'. \quad \begin{pmatrix} \Psi' \vdash e' \rightsquigarrow^{\infty} v' : \tau' \\ v \ \sqsubseteq \ v' \end{pmatrix} \text{ or } \Psi \vdash v \ \texttt{bad-cast}$$

Proof. By Lemma B.19, $\Psi \vdash e \rightsquigarrow^{\infty} v : \tau$ implies either v is a bad cast or $\Psi \vdash e \rightarrow^{\infty} v : \tau$ holds. In the former case, we are done. In the latter case, Corollary B.2 implies either v is a bad cast or $\Psi' \vdash e' \rightarrow^{\infty} v : \tau$ holds. Again, in the former case we are done. In the latter case, Theorem 5.6.3 implies $\Psi' \vdash e' \rightsquigarrow^{\infty} v : \tau$ holds. Applying Lemma B.24 and the assumption that $\tau \sqsubseteq \tau'$ holds, one can easily show this achieves our goal. □

B.4.4 *Gradual Reflection*

Lemma B.27.

$$\forall e, v. \quad e \sqsubseteq v \quad \implies \quad e = v$$

Proof. By induction on the proof of $e \sqsubseteq v$. □

Lemma B.28. *For every Ψ and Ψ' where $\Psi \sqsubseteq \Psi'$ holds, and for all e_1, e_1', and e_2',*

$$e_1 \sqsubseteq e_1' \ \wedge \ \Psi' \vdash e_1' \ R \ e_2' \quad \Longrightarrow \quad \begin{array}{c} \exists e_2. \quad \Psi \vdash e_1 \ R \ e_2 \ \wedge \ e_2 \sqsubseteq e_2' \\ \textit{or} \\ \Psi \vdash e_1 \ \texttt{bad-cast} \\ \textit{or} \\ \forall \tau. \quad \Psi \vdash e_1 \ \texttt{lapse} \ \tau \end{array}$$

Proof. By induction on the proof of $\Psi \vdash e_1' \ R \ e_2'$, applying Lemma B.27 to get values, and applying Lemma B.23 in cases with variable substitutions. □

Lemma B.29. *For every Ψ and Ψ' where $\Psi \sqsubseteq \Psi$ holds,*

$$\forall \varepsilon, e, e'. \quad \begin{pmatrix} e \sqsubseteq e' \\ \Psi' \vdash e' \ \texttt{erroneous} \ \varepsilon \end{pmatrix} \quad \Longrightarrow \quad \begin{array}{c} \Psi \vdash e \ \texttt{erroneous} \ \varepsilon \\ \textit{or} \\ \forall \tau. \quad \Psi \vdash e \ \texttt{lapse} \ \tau \end{array}$$

Proof. By induction on the proof of $\Psi' \vdash e' \ \texttt{erroneous} \ \varepsilon$, applying Lemma B.27 to get values. □

Lemma B.30. *For every Ψ and Ψ' where $\Psi \sqsubseteq \Psi'$ holds, and for all e, e', v', and τ',*

$$\begin{pmatrix} e \sqsubseteq e' \\ \Psi' \vdash e' \twoheadrightarrow^{\infty} v' : \tau' \end{pmatrix} \Longrightarrow \begin{array}{c} \Psi \vdash e \twoheadrightarrow^{\infty} v' : \tau' \\ \textit{or} \\ \exists \varepsilon. \quad \Psi \vdash \varepsilon \ \texttt{bad-cast} \ \wedge \ \forall \tau. \quad \Psi \vdash e \twoheadrightarrow^{\infty} \varepsilon : \tau \\ \textit{or} \\ \forall \tau. \quad \Psi \vdash e \twoheadrightarrow^{*} \texttt{lapse} \ \tau \end{array}$$

Proof. In the case where v' is ∞, Lemma B.28 implies either $\Psi \vdash e \twoheadrightarrow^{\infty} \infty : \tau'$ or e optimistically reduces to some bad cast or a lapse of any type τ. Lemma B.18 then implies that either those optimistic reductions are also pessimistic reductions or e pessimistically reduces to a lapse of any type τ. Any of the resulting cases achieve our goal.

In the case where v' is some value v of type τ', Lemmas B.28 and B.27 imply either $\Psi \vdash e \twoheadrightarrow^{\infty} v : \tau'$ or e optimistically reduces to some bad cast or a lapse of any type τ. Lemma B.18 then implies that either those optimistic reductions are also pessimistic reductions or e pessimistically reduces to a lapse of any type τ. Any of the resulting cases achieve our goal.

In the case where v' is some error ε, Lemmas B.28 and B.29 imply either $\Psi \vdash e \twoheadrightarrow^{\infty} \varepsilon : \tau'$ or e optimistically reduces to some bad cast or a lapse of any type τ. Lemma B.18 then implies that either those optimistic reductions are also pessimistic reductions or e pessimistically reduces to a lapse of any type τ. Any of the resulting cases achieve our goal. □

Lemma B.31. *For every* Ψ *and* Ψ' *where* $\Psi \sqsubseteq \Psi'$ *holds,*

$$\forall e, e', \tau, \tau'. \quad e \sqsubseteq e' \ \wedge\ \tau \sqsubseteq \tau' \ \wedge\ \Psi' \vdash e' \ \mathbf{lapse}\ \tau' \quad \Longrightarrow \quad \Psi \vdash e \ \mathbf{lapse}\ \tau$$

Proof. By induction on the proof of $\Psi' \vdash e' \ \mathbf{lapse}\ \tau'$. □

Theorem 5.7.4 (Gradual Reflection). *For every* Ψ, Ψ', e, e', τ, *and* τ' *such that* $\vdash \Psi$, $\vdash \Psi'$, $\Psi \vdash \tau$, $\Psi' \vdash \tau'$, $\Psi \vdash e \triangleleft \tau$, *and* $\Psi' \vdash e' \triangleleft \tau'$ *hold,*

$$\forall \nu'. \left(\begin{array}{c} \Psi \sqsubseteq \Psi' \\ \tau \sqsubseteq \tau' \\ e \sqsubseteq e' \\ \Psi' \vdash e' \rightsquigarrow^{\infty} \nu' : \tau' \end{array} \right) \Longrightarrow \exists \nu.\ \Psi \vdash e \rightsquigarrow^{\infty} \nu : \tau \wedge \begin{array}{c} \nu \sqsubseteq \nu' \\ \textit{or} \\ \Psi \vdash \nu\ \texttt{bad-cast} \end{array}$$

Proof. By Theorem 5.6.4, $\Psi' \vdash e' \rightsquigarrow^{\infty} \nu' : \tau'$ implies either $\Psi' \vdash e' \rightarrow\!\triangleright^{*}$ `lapse` τ' or $\Psi' \vdash e' \rightarrow\!\triangleright^{\infty} \nu' : \tau'$ hold. In the former case, Lemma B.28 followed by Lemmas B.18 and B.31 imply e pessimistically reduces to some bad cast or to a lapse of type τ. In the latter case, Lemma B.30 implies $\Psi \vdash e \rightarrow^{\infty} \nu' : \tau'$ holds or e pessimistically reduces to some bad cast or to a lapse of type τ. Altogether, this leaves us with three cases to consider. In the case where e pessimistically reduces to some bad cast, this achieves our goal, so we consider the other two cases.

Suppose e pessimistically reduces to a lapse of type τ. Theorem 5.7.1 implies any pessimistically typed refinement of e with respect to τ will optimistically reduce to some bad cast. By definition, this means $\Psi \vdash e \rightsquigarrow^{\infty} \nu : \tau$ holds for some bad cast ν, achieving our goal. Note that Theorem 5.7.1 is stronger than necessary for this. In particular, we do not need to know that a bad cast will be identified immediately if reduction of the original program becomes optimistically ill-typed; we only need to know that one will be identified eventually.

Suppose instead that $\Psi \vdash e \rightarrow^{\infty} \nu' : \tau'$ holds. In the case where ν' is ∞ or some error ε, then $\Psi \vdash e \rightarrow^{\infty} \nu' : \tau$ holds as well, achieving our goal. In the case where ν' is some value v, then either v has type τ or not.

If it does, then $\Psi \vdash e \twoheadrightarrow^{\infty} v' : \tau$ holds, achieving our goal. If it does not, then $\Psi \vdash v$ **lapse** τ holds, implying e pessimistically reduces to a lapse of type τ. As above, we can then apply Thoerem 5.7.1 to show that $\Psi \vdash e \rightsquigarrow^{\infty} v : \tau$ holds for some bad cast v, achieving our goal. □

C

MORE ON MONNOM

C.1 BENCHMARK RESULT CHARTS

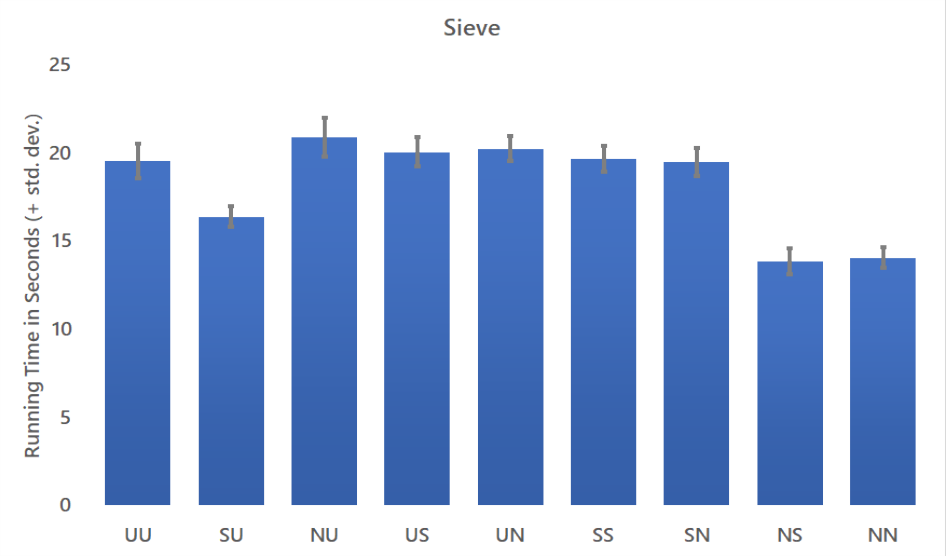


Figure C.1: Bar Graph for MonNom Sieve Results. Versions are labeled *U* (Untyped Structural), *S* (Typed Structural), or *N* (Nominal), and ordered alphabetically (`Main`, `Stream`).; `Intersort`: `Iterator`, `List`, `ListNode`).

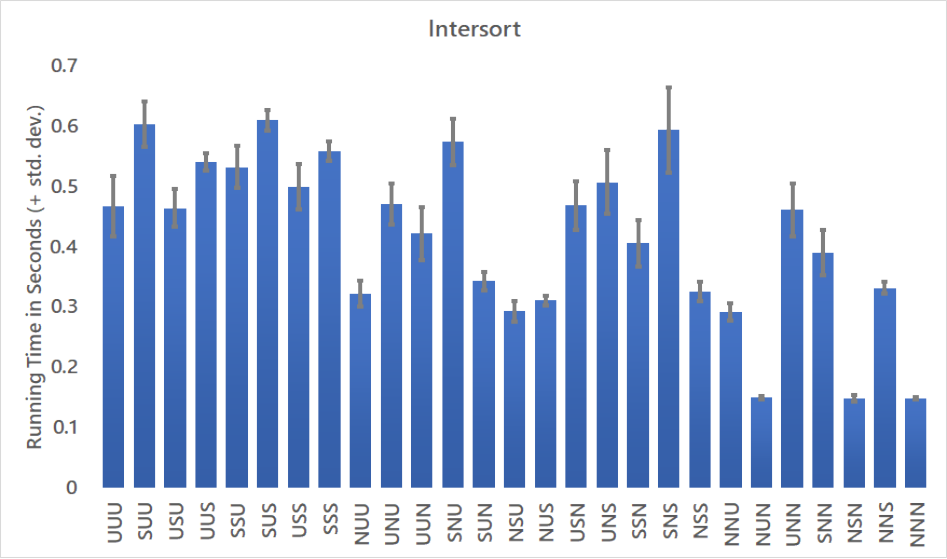


Figure C.2: Bar Graph for MonNom Intersort Results. Versions are labeled *U* (Untyped Structural), *S* (Typed Structural), or *N* (Nominal), and ordered alphabetically (`Iterator`, `List`, `ListNode`).

D

MORE ON GENERICS

D.1 CORPUS STUDY

As discussed in Section 8.2, the current major object-oriented programming languages use type-argument inference algorithms in a somewhat ad-hoc fashion, and their libraries are designed accordingly. However, they heavily use generics to provide rich APIs in particular for their Collections-related libraries, establishing a good baseline for the kinds of methods a reasonable system needs to support. So while back-patching our requirements for inferable type parameters – which include things like `null` not being a member of every type and principal instantiation inheritance – onto these existing languages is infeasible, we can nonetheless analyze their APIs and try to translate them into our System, if necessary with some minor modifications.

In this section, we therefore analyze the Collections-related libraries of C# [ECMA TC39-TG2, 2017], Ceylon [King, 2013], Java [Gosling, Joy, Steele, Bracha, and Buckley, 2015], and Kotlin [JetBrains, 2019], which are usually the main use case of generics. All of those languages feature and heavily use multiple interface inheritance, so we analyzed them by translating the relevant method signatures to $Gen_{\cap}$ as closely as possible. Wherever plausible, we translated function/delegate types to higher-order parameters.

Each of the following subsections lists what namespaces/libraries and what kind of members of those were analyzed, along with statistics about the numbers of inferable parameters with and without simple redesigns, and elaborations on a few of the more interesting cases. Inferable parameters are split up by the variance with which they can be inferred, where $\pm$ indicates that a parameter can be inferred either co- or contravariantly. While this may not make a difference to the inference algorithm, it does make a difference for the code that uses the inferred type parameters; in fact, for some methods, inferring the type parameters at all would be unwise even though it is possible (see an example below).

D.1.1 *C#*

The following tables shows the results for the libraries `System.Linq`, `System.Collections.Immutable`, and a slightly redesigned version of the latter, once grouped by individual type parameters and once grouped by the methods that they are parameters for. The various kinds of inferability refer to the variances with which the type parameters are inferred, where $\pm$ means a parameter could be inferred either co- or contravariantly.

Library	**#**	**Inferable**				**Uninferable**
		+	**-**	**!**	$\pm$	
System.Linq	293	218	2	8	32	33
System.Collections.Immutable	272	22	0	101	33	116
System.Collections.Immutable^R	272	163	0	33	56	20

Table D.1: Inferability of Type Parameters in C# Collections

Library	#	Inferable		Uninferable
		Fully	Partially	
System.Linq	204	182	9	13
System.Collections.Immutable	195	115	14	66
System.Collections.Immutable^R	195	181	7	7

Table D.2: Inferability of Method Signatures in C# Collections

D.1.1.1 *System.Linq*

This library contains most extension methods for `IEnumerable<+T>`, most importantly C#'s version of map (`Select`), fold (`Aggregate`), and filter (`Where`). The only methods with uninferable type parameters in `System.Linq` are various overloadings of `Aggregate`, `ToArray`, `ToList`, `ToLookup`, `ToHashSet`, and `ToDictionary`. The first, `Aggregate`, is uninferable because the aggregator function (in its general form of type $(\alpha, \beta) \to \alpha$) should be a higher-order parameter, which leads to a dependency cycle, as it requires α to be inferred covariantly to have the best input types, but in order for α to be inferred covariantly, the output type of the function, which depends on the input types, would have to already be known. All other methods with uninferable type parameters return instances of invariant classes for good reason, so given that they only have covariant input, there is no way to infer the type arguments for these methods in a principled way. However, in particular uses of `ToArray` and `ToList` are often called just to force the evaluation of otherwise lazily evaluated other extension methods of `IEnumerable`. For these use cases, additional extension methods like `ToReadOnlyList` (whose result would be covariant) would suffice, and their type arguments would be inferable.

There are a few methods whose parameters are technically inferable, but that would be ill-advised. For example, consider the extension method `IEnumerable<T>.Cast<S>() : IEnumerable<S>`. The type parameter `S` is inferable covariantly, and because there is nothing to constrain it, it will always be inferred to be $\bot$. The purpose of the method is to return an enumerable where all the elements were cast to `S`, in this case $\bot$, which will always fail.

D.1.1.2 *System.Collections.Immutable*

This library has a lot more uninferable type parameters. The reason for this is that all the immutable versions of data structures `ImmutableArray`, `ImmutableDictionary`, etc. are invariant, as they are classes and also inherit from invariant interfaces. This is unnecessary, as their implementation of relevant mutating operations throw exceptions (i.e., they offer methods like `void Add(E elem)`); instead, they provide functional versions of those mutating operations (e.g. `ImmutableList<E> Add(E elem)`). In our calculus, all of this is compatible with making the type parameter of the relevant classes/interfaces covariant. Thus, the re-designed version of `System.Collections.Immutable`, where the type parameter declarations on the immutable data structures were changed from invariant to covariant moves most of the previously uninferable type parameters into the covariant column (along with most of the previously uniquely determined, invariant type parameters).

D.1.2 *Ceylon*

Ceylon's collection libraries were built with generics in mind, and thus is the closest match to our calculus. The package `ceylon.collections` itself is fairly sparse, only containing eight functions at the top level. The package `ceylon.language` contains a lot more top-level function, many if which are related to collections, in addition to some higher-order functions like currying. For the latter, Ceylon supports type-level tuples and what is essentially a strong form of variadic type arguments. Our calculus does not have a direct equivalent, so for the below results we assumed that a list of type parameters was just a single parameter; the results generalize for any amount of type parameters positioned and constrained in the same way. Unlike C# and Kotlin, Ceylon specifies the basic list operations like filter, fold, and map directly in the `Iterable` interface, which is therefore also included here.

Namespace	**#**	**Inferable**				**Uninferable**
		+	**-**	**!**	$\pm$	
ceylon.language	109	86	16	0	1	6
ceylon.collection	11	8	0	0	0	3
ceylon.language.Iterable	21	14	1	0	0	5

Table D.3: Inferability of Type Parameters in Ceylon Collections

In the libraries analyzed here, only a few parameters would not be inferable based on our calculus. The reasons for that fall into one of three categories:

- fold methods where, as in the other languages, the type parameter representing the accumulator appears on both sides of a higher-order

Namespace	#	Inferable		Uninferable
		Fully	Partially	
ceylon.language	59	51	3	5
ceylon.collection	8	6	0	2
ceylon.language.Iterable	17	12	0	5

Table D.4: Inferability of Method Signatures in Ceylon Collections

parameter. A special case here is the method `set<E <: Object>(stream : Iterable<E, Null>, choosing(E earlier, E later) : E) : Set<E>`
This method goes through `stream` and creates a set, using the method `choosing` to disambiguate which of two objects to put in the set if two or more equal values are in `stream`. Treating `choosing` as a higher-order parameter means that the type parameter `E` cannot be inferred.

- Returning invariant types based on covariant collection values
- Otherwise covariantly inferable type parameters occurring in union types. Two (`Iterable.group` and `Iterable.summarize`) of the three occurrences of this are caused by nullable types, which in Ceylon mean `E ∪ Null`. Our calculus will treat `E` as not inferable if a parameter has the above type. However, a slight extension of our calculus based on more elaborate disjointness reasoning could infer `E` covariantly in the two methods here, as they are also restricted two be subtypes of `Object`, creating a disjoint union.

D.1.3 *Java*

In Java, we focus on translating the methods in the java.util.Collections interface as well as the java.util.Stream interface, which contains the more

modern higher-order facilities like map, filter, and fold. The following tables show the results. As Java does not feature declaration-site variance, we made some assumptions about some of the most basic interfaces being covariant:

- `Enumerable<+T>`
- `Stream<+T>`
- `Iterator<+T>`

As discussed below, it makes sense for a number of other types to be covariant, altough that would require some minor redesign work - represented in Tables D.5 and D.6 as java.util.Collections^R.

Namespace	**#**	**Inferable**				**Uninferable**
		+	**-**	**!**	$\pm$	
java.util.Collections	65	8	2	31	3	21
java.util.Collections^R	65	29	2	31	3	0
java.util.Stream	15	10	0	2	1	2

Table D.5: Inferability of Type Parameters in Java Collections

Namespace	**#**	**Inferable**		**Uninferable**
		Fully	**Partially**	
java.util.Collections	55	36	0	19
java.util.Collections^R	55	55	0	0
java.util.Stream	14	12	0	2

Table D.6: Inferability of Method Signatures in Java Collections

The biggest source of uninferability in java.util.Collections is that many methods in specifically return immutable objects, yet are treated as being invariant. Thus, we can imagine redesigning the system such that these

immutable objects are treated covariantly as in a setting with declaration-site variance, which would allow our system to successfully infer the corresponding type arguments. As an example, the method `<T> List<T> nCopies(int n, T o)` is not inferable since the return type is invariant in `T` while the argument is covariant in `T`. However, this method's specification stipulates that the returned list is immutable; thus, we could redesign the system to return an object of a type representing immutable lists, which could then be covariant, allowing the method to be inferable with `T` covariant. This redesign of java.util.Collections leave no type parameters uninferable.

Finally, the `Stream<T>` interface has similarities to collections interfaces in other languages and was thus included. By treating `Stream<T>` covariantly, we are able to infer most methods in the class. The one method not inferable is `<T> Stream.Builder<T> builder()`, since making the builder covariant would not be in the spirit of the builder pattern.

D.1.4 *Kotlin*

Kotlin, like C#, implements most collection functionality in extension methods, which are specified in `kotlin.collections`. There were two small groups of uninferable parameters that could have been inferable if two orthogonal issues (one for each group) would be solved. One is a simple re-design of the `Map` interface to be covariant in both parameters (redesign "R"), and the other is an extension of our calculus to allow extended reasoning about union types (redesign "?"). Tables D.7 and D.8

show the original results plus both redesigns and a combination of the two.

Namespace	**#**	**Inferable**				**Uninferable**
		+	**-**	**!**	$\pm$	
kotlin.collections	244	188	0	4	27	25
kotlin.collections^R	244	194	0	4	27	19
kotlin.collections$^?$	244	194	0	4	27	19
kotlin.collections^R?	244	200	0	4	27	13

Table D.7: Inferability of Type Parameters in Kotlin Collections

Namespace	**#**	**Inferable**		**Uninferable**
		Fully	**Partially**	
kotlin.collections	172	147	21	4
kotlin.collections^R	172	153	15	4
kotlin.collections$^?$	172	153	15	4
kotlin.collections^R?	172	159	9	4

Table D.8: Inferability of Method Signatures in Kotlin Collections

D.2 CASE STUDY

In this section, we sketch a standard library that is built using our calculus, demonstrating some of its more advanced features.

D.2.1 *Extensions*

We assume a few extensions to our calculus to make it more convenient to use, most importantly mutable fields.

D.2.1.1 *Inheritance*

For simplicity, the formalization in the chapter demands that each interface/shape explicitly declares all of its methods, which are then checked for whether they satisfy the requirements of the declared super-interfaces/shapes. Similarly, the conditionally satisfied shapes have to be re-declared for each interface. Here, we assume that method declarations and conditionally satisfied shapes can be inherited and would only have to be explicitly declared in cases where two non-identical specifications of the same thing would be inherited.

D.2.1.2 *Mutable Fields*

The formalization in this dissertation passes around immutable object values. We assume a straightforward extension of this where instead, everything is a reference to a heap-allocated object, and objects can have mutable fields that are only accessible from within them. The key point of the accessibility restriction is to allow field types to refer to co- and contravariant type variables - while the variable is effectively invariant within the object itself, an outside view of the object may use a super- or subtype of the actual type argument, respectively, and thus it would be unsound to write to or read from the field, respectively. While this can easily be relaxed, for our purposes, getter- and setter-methods can provide outside access to fields as long as they respect co/contravariance.

There are two new kinds of expressions: value field assignments $x := e; e$, where the first expression evaluates to the value assigned to the field x (same namespace as local variables for convenience, but only fields are mutable) and the second expression evaluates to the result of the expres-

sion as a whole, and function field assignments $f := f; e$, which assigns the value of a function field or a function variable to the specified function field (again same namespace for convenience, but function variables are immutable) and returns the result of the second expression. Variable expressions double as potential read expressions; these variables are only substituted with their value as needed, while all other (immutable) variables are still substituted globally. Furthermore, a declaration of an object now has added field declarations and initialization expressions:

$$\textbf{object}\ x : I\langle\vec{\tau}\rangle\ \{p := e; ...; d; ...\}$$

As stated above, the mutable fields of an object are only accessible from within that object. With respect to that, the field initialization expressions count as being within the object whose code is creating the new object and can thus access the fields of the outer object, but not those of the one being created, while the expressions in method definitions cannot access fields of the outer object, only those of the object being created.

D.2.1.3 *Static/Top-Level Methods/Fields*

Since a program is just a number of interface/shape definitions and an expression, that expression can be wrapped in an initial let-expression that defines a special object (e.g. "Static") that defines the top-level methods and contains the top-level fields and corresponding getters and setters, along with a special interface that specifies its signature. This also provides an easy way to create constructors for "Classes" (see our object "New" below), which are just static methods that return newly constructed objects.

D.2.1.4 *Mixed Named and Positional Arguments*

The calculus uses explicitly named arguments everywhere, but that is not a strict requirement. In the code below, we simply use positional arguments as one would expect.

D.2.2 *Library*

D.2.2.1 *Shapes*

The library starts with a collection of standard shapes, most notably `Eq<-E>`, which forms the top of a hierarchy of types related to comparability. The library also contains two different shapes for clonability - one for deep and one for shallow copies. Lastly, we provide some binary operator shapes for standard operations.

```
// Shape for types that can be equated
shape Eq<-E> {
  Equals(than : E) : Boolean;
}

// Shape for types where equal instances have the same hash
shape HashEq<-E> extends Eq<E> {
  Hash() : Nat;
}

// Shape for partially ordered types
shape PComparable<-E> extends Eq<E> {
  IsLessThan(that : E) : Boolean;
  Compare(to : E) : PComparison;
}
```

```
// Shape for totally ordered types
shape Comparable<-E> extends PComparable<E> {
  Compare(to : E) : Comparison;
}

// Shape for types whose values have joins
shape Joinable<!E> extends PComparable<E> {
  Join(with : E) : E;
}

// Shape for types whose values have meets
shape Meetable<!E> extends PComparable<E> {
  Meet(with : E) : E;
}

// Shape for shallowly clonable types
shape SCloneable<+E> {
  SClone() : E;
}

// Shape for deeply clonable types
shape DCloneable<+E> {
  DClone() : E;
}

// Shape for summable types
shape Summable<!E> {
  Add(right : E) : E;
}

// Shape for summable types whose addition is commutative
shape CommSummable<!E> extends Summable<E> {
```

```
}

// Shape for subtractable types
shape Subtractable<!E> extends CommSummable<E> {
  Subtract(subtrahend : E) : E;
}

// Shape for multipliable types
shape Multipliable<!E> {
  Multiply(right : E) : E;
}

// Shape for divisible types
shape Divisible<!E> extends Multipliable<E> {
  Divide(divisor : E) : E;
}

// Shape for multipliable types whose multiplication is commutative
shape CommMultipliable<!E> extends Multipliable<E> {
}

// Shape for exponentiable types
shape Exponentiable<!E> extends Multipliable<E> {
  Exponentiate(exponent : E) : E;
}

// Short-hand shape for primitive types
shape Primitive<!E> extends Joinable<E>, Meetable<E>, SCloneable<Unit>,
    DCloneable<E> {
}
```

D.2.2.2 *Basic Types*

Next we specify some of the basic types in the library. Note that while we are specifying even the most basic types as interfaces, a more practical version of the language can easily incorporate optimized primitives. The only method here whose type parameters are not inferable is `Number.Repeat`, since `T` occurs covariantly in both the arguments and the return type of `step`. Various versions of case/if-then-else on the other hand are completely inferable.

```
// Unit
interface Unit satisfies Primitive<Unit> {
}

// Boolean
interface Boolean satisfies Primitive<Boolean> {
  IfThenElse[+T](thencont() : T, elsecont() : T);
}

// Numbers
interface Number satisfies Primitive<Number>, Subtractable<Number>,
    Divisible<Number>, CommMultipliable<Number>, Exponentiable<Number> {
}

// Natural Numbers
interface Nat extends Number satisfies DCloneable<Nat>[][],
    SCloneable<Nat>[][] {
  Repeat<T>(init : T, step(T) : T);
}

// Tagged Sum
```

```
interface Either<+T, +S> satisfies DCloneable<Either<V, W>>[T <: +V, S <:
    +W][T.DCloneable<V>, S.DCloneable<W>], SCloneable<Either<T,S>>[][]
{
  Bind[+V,+W](left(T) : V, right(S) : W) : Either<V, W>;
  Case[+V](left(T) : V, right(S) : V) : V;
}

// Option
interface Option<+T> extends Either<T, Unit> satisfies
    DCloneable<Option<S>>[T <: +S][T.Cloneable<S>],
    SCloneable<Option<S>>[][]  {
  BindOption[+S](some(T) : S) : Option<S>;
  CaseOption[+S](some(T) : S, none() : S) : S;
}

// Pair
interface Pair<+T, +S> satisfies Eq<Pair<V, W>>[T <: -V, S <: -W][T.Eq<V>,
    S.Eq<W>], HashEq<Pair<V, W>>[T <: -V, S <: -W][T.HashEq<V>,
    S.HashEq<W>], SCloneable<Pair<T,S>>[][] {
  Left() : T;
  Right() : S;
}

// Function
interface Function<-I, +O> {
  Invoke(i : I) : O;
}

// Result type for comparisons of PComparable
interface PComparison {
  IsLessThan() : Boolean;
  IsEqual() : Boolean;
  IsGreaterThan() : Boolean;
```

```
  IsIncomparable() : Boolean;
  Case[+T](lessThan() : T, equalTo() : T, greaterThan() : T, incomparable()
    : T) : T;
}

// Result type for comparisons of Comparable
interface Comparison {
  Case[+T](lessThan() : T, equalTo() : T, greaterThan() : T) : T;
}
```

D.2.2.3 *Collections*

Collections are one of the main use-cases of generics. An important design principle here was to keep the interface hierarchy covariant for as long as possibe - since our calculus support lower bounds on type variables, we can use them to get around many of the limiations that covariance would usually come with. Note the heavy use of shape evidence to conditionally provide methods depending on the generic arguments in especially `Collection`. `ImmutableLinkedList` demonstrates the principle behind the redesign of the C# System.Collections.Immutable library.

```
// Iterator
interface Iterator<+E> {
  MoveNext() : Boolean;
  Current() : E;
}

// Enumerable
interface Enumerable<+E> {
  IsEmpty() : Boolean;
  GetIterator() : Iterator<E>;
  FoldShortcut<T>[+S](start : T, step(T, E) : Either<T, S>) : Either<T, S>;
```

```
}

// Collection
interface Collection<+E> extends Enumerable<E> {
  Fold<T>(start : T, step(T, E) : T) : T;
  FoldFirst<T>[+R](first(E) : T, empty() : R, step(T, E) : T, result(T) : R)
    : R ;
  Count() : Nat;
  CountOf[E <: +T](elem : T)[E.Eq<T>] : Nat;
  Contains[E <: +T](elem : T)[E.Eq<T>] : Boolean;
  Equals[E <: +T](Collection<T> other)[E.Eq<T>] : Boolean;
  Hash()[E.HashEq<E>] : Nat;
  Sum[E <: !T](start : T)[T.Summable<T>] : T;
  Product[E <: !T](start : T)[T.Multipliable<T>] : T;
  Join[E <: !T](start : T)[T.Joinable<T>] : T;
  Meet[E <: !T](start : T)[T.Meetable<T>] : T;
}

// Set
interface Set<+E> extends Collection<E> satisfies
    Eq<Set<T>>[E<:-T][E.Eq<T>], HashEq<Set<T>>[E<:-T][E.HashEq<T>] { ... }

// Multiset
interface Multiset<+E> extends Collection<Pair<E, Nat>>
    Eq<Multiset<T>>[E<:-T][E.Eq<T>],
    HashEq<Multiset<T>>[E<:-T][E.HashEq<T>] { ... }

// Sequence
interface Sequence<+E> extends Collection<E> satisfies
    Eq<Sequence<T>>[E<:-T][E.Eq<T>],
    HashEq<Sequence<T>>[E<:-T][E.HashEq<T>] {
  First() : E;
  Last() : E;
```

```
  ElementAt(index : Nat) : E;
  IndexOf[E <: +T](T elem)[E.Eq<T>] : Option<Nat>;
  LastIndexOf[E <: +T](T elem)[E.Eq<T>] : Option<Nat>;
}

// Map
interface Map<+K, +E> {
  ContainsKey[K <: +T](T key)[K.Eq<T>] : Boolean;
  Lookup[K <: +T](T key)[K.Eq<T>] : Option<E>;
}

// FiniteMap
interface FiniteMap<+K, +E> extends Collection<Pair<K,E>>, Map<K,E> {
  GetKeys() : Collection<K>;
  GetValues() : Collection<E>;
}

// ImmutableLinkedList
interface ImmutableLinkedList<+E> extends Sequence<E> {
  Cons[E <: +T](elem : T) : ImmutableLinkedList<T>;
  Insert[E <: +T](index : Nat, elem : T) : ImmutableLinkedList<T>;
  Remove[E <: +T](elem : T)[E.Eq<T>] : ImmutableLinkedList<E>;
  RemoveAll(where(E) : Boolean) : ImmutableLinkedList<E>;
  RemoveAt(index : Nat) : ImmutableLinkedList<E>;
  Replace[E <: +T, E <: +S](oldElem : T, newElem : S)[E.Eq<T>] :
    ImmutableLinkedList<S>;
  SetElementAt[E <: +T](index : Nat, elem : T) : ImmutableLinkedList<T>;
}

// ImmutableMap
interface ImmutableMap<+K, +E> extends Map<K,E> {
  MapTo[K <: +T, E <: +S](key : T, elem: S) : ImmutableMap<T,S>;
  ...
```

```
}

// MutableSet
interface MutableSet<!E> extends Set<E> {
  Add(elem : E) : Unit;
  Remove(elem : E) : Unit;
  ...
}

// MutableList
interface MutableList<!E> extends Sequence<E> satisfies
    DCloneable<MutableList<E>>[][E.DCloneable<E>],
    SCloneable<MutableList<E>>[][]{
  Add(elem : E) : Unit;
  Insert(index : Nat, elem : E) : Unit;
  Remove(elem : E)[E.Eq<E>] : Unit;
  RemoveAll(where(E) : Boolean) : Unit;
  RemoveAt(index : Nat) : Unit;
  Replace(oldElem : E, newElem : E)[E.Eq<E>] : Unit;
  Splice(index : Nat, count : Nat, elems : Enumerable<E>) : Unit;
}
```

D.2.2.4 *Implementation*

We wrap any program in a let-expression that defines the New-object, which contains the various constructors for our types. We start with the constructors for `ImmutableLinkedList`, which are straightforward, providing the cases for nil and cons. Next, the constructors for sets construct various implementations for sets; the interesting case is `BitSet`, as already discussed in Section 8.5.4, the signature of its `Contains` method can be written a lot simpler than its original specification in `Collection` would have suggested.

Last, there are three kinds of examples of how dynamic satisfaction and subtyping checks on types can help implement several optimized classes while still maintaining predictable semantics.

1. `Set()` and `Map()` reason dynamically about the given argument type to deduce the most fitting implementation of a set or map depending on the kinds of equalities or comparisons the given type supports.

2. `Memoize()` takes a higher-order parameter and wraps it in a Function interface. In addition, if the argument type supports at least basic equality, some form of Map (chosen by whatever equalities or comparisons are supported) will remember all computed values and try to look them up instead of calling the function a second time.

3. `SummedList()` creates a list of `Summable` objects. By `Collection`'s interface definition, that means that `SummedList()` has a `Sum()` function that is supposed to return the sum of all its elements. This might be an expensive operation to do all the time, so `SummedList` tries to keep track of what the sum should be in an extra field. The value is first generated when `Sum()` is called the first time, and will be updated when values are added or removed to the list so it stays up-to-date. There's a catch here: if the elements of the list can be added up, but addition is not commutative, then we cannot correctly update our memorized sum value when inserting an element into the middle of the list by just adding its value to our sum value. In this case, we have to erase the memorized sum value and re-compute it next time `Sum()` is called. However, if addition is commutative, that is, if the type argument satisfies `CommSummable`, then we can just add the value to our memorized sum value and do not need to completely recompute it.

Similarly, removing an element while keeping the memorized sum up-to-date requires that the element type be `Subtractable`. All of this can be expressed in our calculus in a type-safe, principled manner.

```
interface New { ... }
let New : New := object New : New {
  ImmutableLinkedList() : ImmutableLinkedList<⊥> ↦
    object this : ImmutableLinkedList<⊥> {
  }
  ImmutableLinkedList[+E](first : E, rest : ImmutableLinkedList<E>) :
     ImmutableLinkedList<E> ↦
    object this : ImmutableLinkedList<E> {
    }
  HashSet<E>()[E.HashEq<E>] : MutableSet<E> ↦
    object this : MutableSet<E> { ... }
  TreeSet<E>()[E.Comparable<E>] : MutableSet<E> ↦
    object this : MutableSet<E> { ... }
  ArraySet<E>()[E.Comparable<E>] : MutableSet<E> ↦
    object this : MutableSet<E> { ... }
  BitSet() : MutableSet<Nat> ↦
    object this : BitSet {
      ...
      // Implements Collection<Nat>.Contains
      Contains(elem : Nat) : Boolean ↦ ...
      ...
    }
  ...
  //Creates a mutable Set that's using the most
  //fitting implementation depending on the level of
  //equality/comparability of the element type.
  Set<E>()[E.Eq<E>] : MutableSet<E> ↦
    if E <: Nat then
      New.BitSet()
```

```
  else
    if E satisfies HashEq<E> then
      New.HashSet<E>()
    else
      if E satisfies Comparable<E> then
        New.TreeSet<E>()
      else
        New.ArraySet<E>();

//Creates a mutable Map that's using the most
//fitting implementation depending on the level of
//equality/comparability of the key type.
Map<K,E>()[K.Eq<K>] : MutableMap<K,E> ↦
  if E satisfies HashEq<E> then
    New.HashMap<K, E>()
  else
    if E satisfies Comparable<E> then
      New.TreeMap<K, E>()
    else
      New.ArrayMap<K, E>();

//Takes a function and creates a memoized version of it
//if the parameter type is at least equatable, otherwise
//just wraps it in a Function object
Memoize<I>[+O](fun(I) : O) : Function<I, O> ↦
  if I satisfies Eq<I> then
    object this : Function<I, O> {
      f(I) : O := fun;
      map : MutableMap<I,O> := New.Map<I,O>();
      Invoke(i : I) : O ↦
        map.Lookup(i).CaseOption(
          (o)↦o,
          ()↦
```

```
            let o := f(i) in
              let x := map.Add(o) in o)
    }
  else
    object this : Function<I, O> {
      f(I) : O := fun;
      Invoke(i : I) : O ↦ f(i)
    };

// Creates a mutable list of summable items that tries to
// keep the current sum in memory to avoid recomputing it
// every time
SummedList<E>()[E.Summable<E>] : MutableList<E> ↦
  object this : MutableList<E> {
    sum : Optional<E> := New.None();
    list : MutableList<E> := New.ArrayList();

    // Returns the sum of all the elements in the list,
    // either from memory or recomputes the sum of all the
    // elements in the list and stores the result in memory
    Sum(start : E) : E ↦
      sum.CaseOption(
        (v) ↦ start.Add(v),
        () ↦
          list.FoldFirst<E>(
            (e) ↦ e,
            () ↦ New.None(),
            (acc, e) ↦ acc.Add(e),
            (r) ↦ New.Some(r)).CaseOption(
              (v) ↦ sum := v; start.Add(v),
              () ↦ start));

    // Called whenever an element is added at the end. Will add
```

```
    // the element's value to current sum, if applicable
    Add(elem : E) : Unit ↦
      let u ≔ list.Add(elem) in
        sum ≔ sum.BindOption((v) ↦ v.Add(elem)); u;

    // Called whenever an element is inserted. Will try to add
    // the element's value to current sum if possible, otherwise
    // resets current sum
    Insert(index : Nat, elem : E) : E ↦
      if E satisfies CommSummable<E> then
        sum ≔ sum.BindOption((v) ↦ v.Add(elem)); list.Insert(index,
  elem);
      else
        sum ≔ New.None(); list.Insert(index, elem);

    // Called whenever an element is removed. Will try to subtract
    // the element's value from current sum if possible, otherwise
    // resets current sum
    Remove(elem : E) : E ↦
      if E satisfies Subtractable<E> then
        sum ≔ sum.BindOption((v) ↦ v.Subtract(elem)); list.Remove(elem);
      else
        sum ≔ New.None(); list.Remove(elem);

    ... // Implementations of all other MutableList<E> methods
  };
} in //[Your Program Here]
```

D.3 FORMALIZATION INDEX

Hierarchy	Ψ	$::= c; ...$
Confluence	c	$::= c^I \mid c^S$
Interface Name	I	
Interface Declaration	c^I	$::= \texttt{interface}\ I\langle\Theta\rangle\ \texttt{extends}\ \vec{\tau}^I\ \texttt{satisfies}\vec{\sigma}^I\{s; ...\}$
Kind Context	Θ	$::= \tau <: \nu\alpha <: \tau, ...$
Inherited Interfaces	$\vec{\tau}^I$	$::= \varnothing \mid \tau^I \mid \tau^I, ...$
Inherited Interface	τ^I	$::= I\langle\alpha, ...\rangle \mid I\langle\vec{\tau}\rangle$
Type Variable	α	
Types	$\vec{\tau}$	$::= \tau, ...$
Type	τ	$::= \bot \mid \top \mid I\langle\vec{\tau}^a\rangle \mid \alpha \mid \tau \cup \tau \mid \tau \cap \tau$
Type Arguments	$\vec{\tau}^a$	$::= \tau^a, ...$
Type Argument	τ^a	$::= \tau \mid \textbf{in}\ \tau\ \textbf{out}\ \tau$
Signed Variance	$\nu^{\pm}$	$::= + \mid -$
Ignorable Variance	$\nu^?$	$::= \nu \mid ?$
Variance	ν	$::= \nu^{\pm} \mid !$
Shape Name	S	
Shape Declaration	c^S	$::= \mathcal{S}S\Theta\sigma, ...s; ...$
Shape	σ	$::= S\langle\vec{\tau}\rangle$
Evidence Variable	ς	
Conditionally Satisfied Shapes	$\vec{\sigma}^I$	$::= \sigma^I, ...$
Conditionally Satisfied Shape	σ^I	$::= \sigma[\Theta][\Sigma]$
Shape Context	Σ	$::= \varsigma : \alpha.\sigma, ...$
Shape Premise	Σ^τ	$::= \varsigma : \tau.\sigma, ...$
Shape Conclusion	$\Sigma^{<:}$	$::= \bot \mid \langle\Theta\rangle[\Sigma]$
Inferred Bounds	$\Theta^{<:}$	$::= \varnothing \mid \Theta^{<:}, \tau <: \alpha \mid \Theta^{<:}, \alpha <: \tau$
Inhabitation	ι	$::= \uparrow \mid \downarrow$

Figure D.1: Full Grammar

Method Name	m	
Program Variable	x	
Function Variable	f	
Method Signature	s	$::= m\langle\Theta^!\rangle[\Theta](\Gamma)[\Sigma] : \tau$
Method Premise	s^τ	$::= m\langle\Theta^!\rangle[\Theta](\Gamma)[\Sigma^\tau] : \tau$
Invariant Kind Context	$\Theta^!$	$::= \tau <: !\alpha <: \tau, \ldots$
Program Context	Γ	$::= p; \ldots$
Program Parameter	p	$::= x : \tau \mid f(\tau, \ldots) : \tau$
Expression	e	$::= x \mid f(e, \ldots) \mid \texttt{let}\ x := e\ \texttt{in}\ e \mid$
		$\mid \texttt{throw} \mid \texttt{if}\ e^g\ \texttt{then}\ e\ \texttt{else}\ e$
		$\mid \texttt{object}\ x : I\langle\vec{\tau}\rangle\ \{d; \ldots\} \mid e^m.m\langle\vec{\tau}\rangle(a; \ldots)$
		$\mid \texttt{capture}\ e\ \texttt{as}\ x : I\langle\vec{\alpha}^!\rangle\ \texttt{in}\ e$
		$\mid \mid \texttt{capture}\ x\ \texttt{as}\ I\langle\vec{\alpha}^!\rangle\ \texttt{in}\ e$
Guard Expression	e^g	$::= \alpha\ \texttt{satisfies}\ \sigma\ \texttt{as}\ \varsigma \mid \tau <: \alpha$
		$\mid \alpha <: \tau \mid e\ \texttt{is}\ x : \tau \mid x\ \texttt{is}\ \tau$
Receiver Expression	e^m	$::= e \mid e@\varsigma$
Program Argument	a	$::= x \mapsto e \mid f(x, \ldots) \mapsto e$
Method Definition	d	$::= s \mapsto e$
Receiver Type	τ^m	$::= \tau \mid \sigma$
Capture Variables	$\vec{\alpha}^!$	$::= \alpha^!, \ldots$
Capture Variable	$\alpha^!$	$::= \alpha \mid \_$
Term	t	$::= x \mid f(t, \ldots) \mid \texttt{let}\ x := t\ \texttt{in}\ t \mid \texttt{throw}$
		$\mid \texttt{if}\ t^g\ \texttt{then}\ t\ \texttt{else}\ t \mid \texttt{object}\ x : I\langle\vec{\tau}\rangle\ \{d^t; \ldots\}$
		$\mid t^m.m\langle\vec{\tau}\rangle(a^t; \ldots) \mid \texttt{capture}\ t\ \texttt{as}\ x : I\langle\vec{\alpha}^!\rangle\ \texttt{in}\ t$
		$\mid \texttt{capture}\ x\ \texttt{as}\ I\langle\vec{\alpha}^!\rangle\ \texttt{in}\ t \mid \texttt{capture}\ v\ \texttt{as}\ I\langle\vec{\alpha}^!\rangle\ \texttt{in}\ t$
Guard Term	t^g	$::= \tau\ \texttt{satisfies}\ \sigma\ \texttt{as}\ \varsigma \mid \tau <: \tau \mid t\ \texttt{is}\ x : \tau$
		$\mid x\ \texttt{is}\ \tau \mid v\ \texttt{is}\ \tau$
Receiver Term	t^m	$::= t \mid t@\varsigma$
Argument Term	a^t	$::= x \mapsto t \mid f(x, \ldots) \mapsto t$
Method Term	d^t	$::= s^\tau \mapsto t$
Value	v	$::= \texttt{object}\ x : I\langle\vec{\tau}\rangle\ \{d^t; \ldots\}$
Argument Value	a^v	$::= x \mapsto v \mid f(x, \ldots) \mapsto t$
Term Context	E	$::= f(v, \ldots, E, t, \ldots) \mid \texttt{let}\ x := E\ \texttt{in}\ t$
		$\mid \mid \texttt{if}\ E\ \texttt{is}\ x : \tau\ \texttt{then}\ t\ \texttt{else}\ t \mid E^m.m\langle\vec{\tau}\rangle(a^t; \ldots)$
		$\mid v.m\langle\vec{\tau}\rangle(a^v; \ldots; x \mapsto E; a^t; \ldots)$
		$\mid \mid \texttt{capture}\ E\ \texttt{as}\ x : I\langle\vec{\alpha}^!\rangle\ \texttt{in}\ t \mid \odot$

Figure D.1 (contd.): Full Grammar (Generics)

Definitions	Figure 8.3: Variance	$\nu * \nu^?$
	Figure 8.4: Type Argument Substitution	$\lfloor \vec{\tau}/\Theta \rceil$
Judgements	Figure 8.1: Programs and Hierarchies	$\vdash \Psi; e$ $\vdash \Psi$
	Figure 8.2: Interfaces	$\vdash^I \Psi$ $\Psi \vdash^I \langle \Theta \rangle$ $\Psi \mid \Psi \mid \Theta \vdash^I \vec{\tau}^I$ $\quad$ $\Psi \mid \Psi \mid \Theta \vdash^I \tau^I$
	Figure 8.3: Variance	$\vdash \nu \leq \nu^?$
	Figure 8.3: Types	$\Psi \mid \Theta \vdash_{\nu^?} \tau$ $\quad$ $\Psi \mid \Theta \vdash_{\nu^?} \tau^I$ $\Psi \mid \Theta \vdash_{\nu^?} \langle \vec{\tau} \rangle : \langle \Theta \rangle$ $\Psi \mid \Theta \vdash \langle \vec{\tau}^a \rangle \rightsquigarrow \langle \vec{\tau} \rangle \mid \Theta^!$ $\vdash \langle \vec{\alpha}^! \rangle := \langle \vec{\tau}^a \rangle \rightsquigarrow \langle \vec{\tau} \rangle \mid \Theta^!$
	Figure 8.3: Subtyping	$\Psi \mid \Theta \vdash \tau <: \tau$ $\quad$ $\Psi \mid \Theta \vdash \tau^I <: \tau^I$ $\Psi \mid \Theta \vdash \langle \vec{\tau} \rangle <: \langle \vec{\tau} \rangle : \langle \Theta \rangle$ $\Psi \mid \Theta \vdash \langle \tau \rangle <: \langle \tau \rangle : \langle \nu \rangle$ $\Psi \mid \Theta \vdash \langle \vec{\tau} \rangle \cap \langle \vec{\tau} \rangle <: \langle \vec{\tau} \rangle : \langle \Theta \rangle$ $\Psi \mid \Theta \vdash \langle \tau \rangle \cap \langle \tau \rangle <: \langle \tau \rangle : \langle \nu \rangle$
	Figure 8.5: Shapes	$\Psi \vdash^S \Psi$ $\Psi \vdash^S \langle \Theta \rangle$ $\Psi \mid \Psi \mid \Theta \vdash^S \sigma, \ldots$ $\Psi \mid \Psi \mid \Theta \vdash^S \sigma, \ldots \sqsubseteq \sigma$ $\Psi \mid \Theta \vdash_{\nu^?} \sigma$ $\Psi \mid \Theta \vdash \sigma \sqsubset: \sigma$
	Figure 8.6: Conditionally Satisfied Shapes	$\Psi \vdash^\sigma \Psi$ $\Psi \mid \Psi \mid \Psi \mid \Theta \vdash^\sigma \vec{\sigma}^I$ $\Psi \mid \Theta \vdash^\sigma \vec{\sigma}^I \sqsubseteq \tau^I$ $\Psi \mid \Theta \mid \Sigma \vdash^\sigma \vec{\sigma}^I \sqsubseteq$ $\Psi \mid \Theta \mid \Sigma \vdash [\Sigma]$

Figure D.2: Static-Formalization Index

Judgements		
	Figure 8.7: Satisfaction	$\Psi \mid \Theta \mid \Sigma \mid \Theta^{<:} \vdash \Sigma^{<:}$ $\Psi \mid \Theta \mid \Sigma \vdash \Sigma^{\tau}$ $\quad$ $\Psi \mid \Theta \mid \Sigma \vdash \tau.\sigma$
	Figure 8.7: Kind Contexts	$\vdash \langle\Theta\rangle \rightsquigarrow \langle\Theta^{!}\rangle$ $\Psi \mid \Theta \vdash \langle\Theta\rangle$ $\Psi \mid \Theta^{<:} \vdash \langle\Theta\rangle \rightsquigarrow \langle\Theta\rangle$
	Figure 8.7: Bound Inference	$\Psi \mid \Theta \vdash \tau <: \tau \rightsquigarrow \langle\Theta^{<:}\rangle$ $\Psi \mid \Theta \vdash \sigma \sqsubset: \sigma \rightsquigarrow \langle\Theta^{<:}\rangle$ $\Psi \mid \Theta \vdash_{I} \tau, \ldots <: \langle\vec{\tau}^{\rangle} \rightsquigarrow \langle\Theta^{<:}\rangle$ $\Psi \mid \Theta \vdash_{\langle\Theta\rangle} \langle\vec{\tau}\rangle, \ldots <: \langle\vec{\tau}^{a}\rangle \rightsquigarrow \langle\Theta^{<:}\rangle$ $\Psi \mid \Theta \vdash_{\nu} \tau, \ldots <: \tau^{a} \rightsquigarrow \langle\Theta^{<:}\rangle$
	Figure 8.8: Inferability	$\Psi \mid \Theta \vdash (\Gamma)[\Sigma] \rightsquigarrow \langle\Theta \mid \Theta^{!}\rangle$ $\Psi \mid \Theta \vdash (\Gamma)[\Sigma] \rightsquigarrow \nu\alpha$ $\Psi \mid \Theta \vdash (p) \rightsquigarrow \nu^{\pm}\alpha$ $\Psi \mid \Theta \vdash \nu\tau^{m}\iota \rightsquigarrow \nu\alpha\iota$ $\Psi \mid \Theta \vdash \nu\langle\vec{\tau}\rangle\iota : \langle\Theta\rangle \rightsquigarrow \nu\alpha$
	Figure 8.8: Method Signatures	$\Psi \vdash^{s} \Psi$ $\Psi \mid \Theta \mid \Sigma \vdash s$ $\Psi \mid \Theta \vdash (\Gamma)$ $\quad$ $\Psi \mid \Theta \vdash (p)$
	Figure 8.8: Inheritance	$\iota, \ldots \vdash \iota$ $\Psi \mid \Theta \mid \Sigma \vdash \{s; \ldots\} \sqsubseteq \tau$ $\quad$ $\Psi \mid \Theta \mid \Sigma \vdash \{s; \ldots\} \sqsubseteq \vec{\tau}^{I}$ $\Psi \mid \Theta \mid \Sigma \vdash \{s; \ldots\} \sqsubseteq \sigma$ $\Psi \mid \Theta \mid \Sigma \vdash \{s; \ldots\} \sqsubseteq s^{\tau}$ $\Psi \mid \Theta \vdash (\Gamma) <: (\Gamma)$ $\quad$ $\Psi \mid \Theta \vdash (p) <: (p)$
	Figure 8.9: Expressions	$\Psi \mid \Theta^{!} \mid \Sigma \mid \Gamma \vdash e : \tau$ $\quad$ $\Psi \mid \Theta^{!} \mid \Sigma \mid \Gamma \vdash e^{m} : \tau^{m}$ $\Psi \mid \Theta^{!} \mid \Sigma \mid \Gamma \vdash (a) : (p)$ $\Psi \mid \Theta^{!} \mid \Sigma \mid \Gamma \vdash d$
	Figure 8.9: Guard Expressions	$\Psi \mid \Theta^{!} \mid \Sigma \mid \Gamma \vdash e^{g} : \Theta^{!} \mid \Sigma \mid \Gamma$
	Figure 8.9: Method Invocation	$\Psi \mid \Theta^{!} \mid \Sigma \vdash \tau^{m}.m\langle\vec{\tau}\rangle(\Gamma) : \tau$ $\Psi \vdash \tau^{m}.s^{\tau}$

Figure D.2 (contd.): Static-Formalization Index

D.4 TERM TYPING AND EQUIVALENCE

Term Typing $\Psi \mid \Theta^! \mid \Sigma^\tau \mid \Gamma \vdash^t t : \tau \quad \Psi \mid \Theta^! \mid \Sigma^\tau \mid \Gamma \vdash^t t^m : \tau^m$

$$\frac{\Psi \mid \Theta^! \mid \Sigma^\tau \mid \Gamma \vdash^t t : \tau \qquad \Psi \mid \Theta^! \vdash \tau <: \tau'}{\Psi \mid \Theta \mid \Sigma^\tau \vdash^t t : \tau'} \qquad \frac{\varsigma : \tau.\sigma \in \Sigma^\tau \qquad \Psi \mid \Theta^! \mid \Sigma^\tau \mid \Gamma \vdash^t t : \tau}{\Psi \mid \Theta^! \mid \Sigma^\tau \mid \Gamma \vdash^t t@_\varsigma : \sigma} \qquad \frac{x : \tau \in \Gamma}{\Psi \mid \Theta^! \mid \Sigma^\tau \mid \Gamma \vdash^t x : \tau}$$

$$\frac{f(\tau_1, \ldots) : \tau \in \Gamma \qquad \Psi \mid \Theta^! \mid \Sigma^\tau \mid \Gamma \vdash^t t_1 : \tau_1 \qquad \ldots}{\Psi \mid \Theta^! \mid \Sigma^\tau \mid \Gamma \vdash^t f(t_1, \ldots) : \tau} \qquad \frac{\Psi \mid \Theta^! \mid \Sigma^\tau \mid \Gamma \vdash^t t_x : \tau_x \qquad \Psi \mid \Theta^! \mid \Sigma^\tau \mid \Gamma; x : \tau_x \vdash^t t : \tau}{\Psi \mid \Theta^! \mid \Sigma^\tau \mid \Gamma \vdash^t \mathbf{let}\ x := t_x\ \mathbf{in}\ t : \tau}$$

$$\frac{}{\Psi \mid \Theta^! \mid \Sigma^\tau \mid \Gamma \vdash \mathbf{throw} : \bot}$$

$$\frac{\Psi \mid \Theta^! \mid \Sigma^\tau \mid \Gamma \vdash^t t^g : \Theta^!_g \mid \Sigma^\tau_g \mid \Gamma_g \qquad \Psi \mid \Theta^!_g \mid \Sigma^\tau_g \mid \Gamma_g \vdash^t t_t \qquad \Psi \mid \Theta^! \mid \Sigma^\tau \mid \Gamma \vdash^t t_f : \tau}{\Psi \mid \Theta^! \mid \Sigma^\tau \mid \Gamma \vdash^t \mathbf{if}\ t^g\ \mathbf{then}\ t_t\ \mathbf{else}\ t_f : \tau}$$

$$\frac{\Psi \mid \Theta^! \vdash_? I\langle\vec{\tau}\rangle \qquad \Psi \mid \Theta^! \mid \Sigma^\tau \mid \Gamma; x : I\langle\vec{\tau}\rangle \vdash^t s_1^\tau \mapsto t_1 \qquad \ldots \qquad \Psi \mid \Theta^! \mid \Sigma^\tau \vdash^t \{s_1^t; \ldots\} \sqsubseteq I\langle\vec{\tau}\rangle}{\Psi \mid \Theta^! \mid \Gamma \mid \Sigma^\tau \vdash^t \mathbf{object}\ x : I\langle\vec{\tau}\rangle\ \{s_1^t \mapsto t_1; \ldots\} : I\langle\vec{\tau}\rangle}$$

$$\frac{\Psi \mid \Theta^! \vdash_? \tau_1 \qquad \ldots \qquad \Psi \mid \Theta^! \mid \Sigma^\tau \mid \Gamma \vdash^t t^m : \tau^m \qquad \Psi \mid \Theta^! \mid \Sigma^\tau \mid \Gamma \vdash^t (a_1^t) : (p_1) \qquad \ldots \qquad \Psi \mid \Theta^! \mid \Sigma^\tau \vdash \tau^m.m\langle\tau_1, \ldots\rangle(p_1; \ldots) : \tau}{\Psi \mid \Theta^! \mid \Sigma^\tau \mid \Gamma \vdash^t t^m.m\langle\tau_1, \ldots\rangle(a_1^t; \ldots) : \tau}$$

$$\frac{\Psi \mid \Theta^! \mid \Sigma^\sigma \mid \Gamma \vdash^t t : I\langle\vec{\tau}^a\rangle \qquad \vdash \langle\vec{\alpha}^!\rangle := \langle\vec{\tau}^a\rangle \rightsquigarrow \langle\vec{\tau}\rangle \mid \Theta^!_\alpha \qquad \Psi \mid \Theta^!, \Theta^!_\alpha \mid \Sigma^\tau \mid \Gamma; x : I\langle\vec{\tau}\rangle \vdash t_x : \tau}{\Psi \mid \Theta^! \mid \Sigma^\tau \mid \Gamma \vdash^t \mathbf{capture}\ t\ \mathbf{as}\ x : I\langle\vec{\alpha}^!\rangle\ \mathbf{in}\ t_x : \tau}$$

$$\frac{\Psi \mid \Theta^! \vdash \tau <: I\langle\vec{\tau}^a\rangle \qquad \vdash \langle\vec{\alpha}^!\rangle := \langle\vec{\tau}^a\rangle \rightsquigarrow \langle\vec{\tau}\rangle \mid \Theta^!_\alpha \qquad \Psi \mid \Theta^!, \Theta^!_\alpha \mid \Sigma^\tau \mid \Gamma_1; x : \tau \cap I\langle\vec{\tau}\rangle; \Gamma_2 \vdash t_x : \tau}{\Psi \mid \Theta^! \mid \Sigma^\tau \mid \Gamma_1; x : \tau; \Gamma_2 \vdash^t \mathbf{capture}\ x\ \mathbf{as}\ I\langle\vec{\alpha}^!\rangle\ \mathbf{in}\ t_x : \tau}$$

$$\frac{\Psi \mid \Theta^! \mid \Sigma^\tau \mid \Gamma \vdash^t t : \tau_1 \qquad \Psi \mid \Theta^! \mid \Sigma^\tau \mid \Gamma \vdash^t t : \tau_2}{\Psi \mid \Theta^! \mid \Sigma^\tau \mid \Gamma \vdash^t t : \tau_1 \cap \tau_2}$$

Figure D.3: Term Typing

Method-Term Typing $\Psi \mid \Theta^! \mid \Sigma^\tau \mid \Gamma \vdash^t d^t$

$$\frac{\Psi \mid \Theta^! \mid \Sigma^\tau \vdash^t m\langle\Theta_e^!\rangle[\Theta_i](\Gamma_m)[\Sigma_m^\tau] : \tau \qquad \vdash \langle\Theta_i\rangle \rightsquigarrow \langle\Theta_i^!\rangle \qquad \Psi \mid \Theta^!, \Theta_e^!, \Theta_i^! \mid \Gamma, \Gamma_m \mid \Sigma^\tau, \Sigma_m^\tau \vdash^t t : \tau}{\Psi \mid \Theta^! \mid \Sigma^\tau \mid \Gamma \vdash^t m\langle\Theta_e^!\rangle[\Theta_i](\Gamma_m)[\Sigma_m^\tau] : \tau \mapsto t}$$

Argument-Term Typing $\Psi \mid \Theta^! \mid \Sigma^\tau \mid \Gamma \vdash^t (a^t) : (p)$

$$\frac{\Psi \mid \Theta^! \mid \Sigma^\tau \mid \Gamma \vdash^t t : \tau}{\Psi \mid \Theta^! \mid \Sigma^\tau \mid \Gamma \vdash^t (x \mapsto t) : (x : \tau)}$$

$$\frac{\Psi \mid \Theta^! \mid \Sigma^\tau \mid \Gamma; x_1 : \tau_1; \ldots \vdash^t t : \tau}{\Psi \mid \Theta^! \mid \Sigma^\tau \mid \Gamma \vdash^t (f(x_1, \ldots) \mapsto t) : (f(\tau_1, \ldots) : \tau)}$$

Guard-Term Typing $\Psi \mid \Theta^! \mid \Sigma^\tau \mid \Gamma \vdash^t t^g : \Theta^! \mid \Sigma^\tau \mid \Gamma$

$$\frac{\Psi \mid \Theta^! \vdash^t [\varsigma : \tau.\sigma]}{\Psi \mid \Theta^! \mid \Sigma^\tau \mid \Gamma \vdash^t \tau \textbf{ satisfies } \sigma \textbf{ as } \varsigma : \Theta^! \mid \Sigma^\tau, \varsigma : \tau.\sigma \mid \Gamma}$$

$$\frac{\Theta^! = \Theta_1^!, \tau_\ell <: !\alpha <: \tau_u, \Theta_2^! \qquad \Psi \mid \Theta_1^! \vdash_? \tau \qquad \Psi \mid \Theta_1^! \vdash \tau_\ell <: \tau \qquad \Psi \mid \Theta_1^! \vdash \tau <: \tau_u}{\Psi \mid \Theta^! \mid \Sigma^\tau \mid \Gamma \vdash^t \tau <: \alpha : \Theta_1^!, \tau <: !\alpha <: \tau_u, \Theta_2^! \mid \Sigma^\tau \mid \Gamma}$$

$$\frac{\Theta^! = \Theta_1^!, \tau_\ell <: !\alpha <: \tau_u, \Theta_2^! \qquad \Psi \mid \Theta_1^! \vdash_? \tau \qquad \Psi \mid \Theta_1^! \vdash \tau_\ell <: \tau \qquad \Psi \mid \Theta_1^! \vdash \tau <: \tau_u}{\Psi \mid \Theta^! \mid \Sigma^\tau \mid \Gamma \vdash^t \alpha <: \tau : \Theta_1^!, \tau_\ell <: !\alpha <: \tau, \Theta_2^! \mid \Sigma^\tau \mid \Gamma}$$

$$\frac{\Psi \mid \Theta^! \vdash_? \tau_\ell \qquad \Psi \mid \Theta^! \vdash_? \tau_u}{\Psi \mid \Theta^! \mid \Sigma^\tau \mid \Gamma \vdash^t \tau_\ell <: \tau_u : \Theta^! \mid \Sigma^\tau \mid \Gamma}$$

$$\frac{\Psi \mid \Theta^! \mid \Sigma^\tau \mid \Gamma \vdash^t t : \top}{\Psi \mid \Theta^! \mid \Sigma^\tau \mid \Gamma \vdash^t t \textbf{ is } x : \tau : \Theta^! \mid \Sigma^\tau \mid \Gamma; x : \tau}$$

$$\frac{}{\Psi \mid \Theta^! \mid \Sigma^\tau \mid \Gamma_1; x : \tau'; \Gamma_2 \vdash x \textbf{ is } \tau : \Theta^! \mid \Sigma^\tau \mid \Gamma_1; x : \tau' \cap \tau; \Gamma_2}$$

$$\frac{\Psi \mid \Theta^! \mid \Sigma^\tau \mid \Gamma \vdash v : \top}{\Psi \mid \Theta^! \mid \Sigma^\tau \mid \Gamma \vdash v \textbf{ is } \tau : \Theta^! \mid \Sigma^\tau \mid \Gamma}$$

Figure D.3 (contd.): Term Typing (Generics)

Term-Invocation Typing $\Psi \mid \Theta^! \mid \Sigma^\tau \vdash^t \tau^m.m\langle\vec{\tau}\rangle(\Gamma) : \tau$

$$\frac{}{\Psi \mid \Theta^! \mid \Sigma^\tau \vdash^t \bot.m\langle\bullet\rangle(\bullet) : \bot} \qquad \frac{}{\Psi \mid \Theta^! \mid \Sigma^\tau \vdash^t \bullet.m\langle\bullet\rangle(...; x : \bot; ...) : \bot}$$

$$\frac{\Psi \mid \Theta^! \mid \Sigma^\tau \vdash^t \tau_m.m\langle\vec{\tau}\rangle(\Gamma) : \tau \qquad \Psi \mid \Theta^! \mid \Sigma^\tau \vdash^t \tau'_m.m\langle\vec{\tau}\rangle(\Gamma) : \tau}{\Psi \mid \Theta^! \mid \Sigma^\tau \vdash^t (\tau_m \cup \tau'_m).m\langle\vec{\tau}\rangle(\Gamma) : \tau}$$

$$\frac{\Psi \mid \Theta^! \mid \Sigma^\tau \vdash^t \tau_m.m\langle\vec{\tau}\rangle(\Gamma) : \tau \qquad \Psi \mid \Theta^! \mid \Sigma^\tau \vdash^t \tau'_m.m\langle\vec{\tau}\rangle(\Gamma) : \tau'}{\Psi \mid \Theta^! \mid \Sigma^\tau \vdash^t (\tau_m \cap \tau'_m).m\langle\vec{\tau}\rangle(\Gamma) : \tau \cap \tau'}$$

$$\frac{\begin{array}{c}\Psi \vdash \tau^m.m\langle\Theta^!_m\rangle[\Theta_m](\Gamma_m)[\Sigma^\tau_m] : \tau \qquad \Psi \mid \Theta^! \vdash_? \langle\vec{\tau}\rangle : \langle\Theta^!_m\rangle \\ \Psi \mid \Theta^! \vdash_? \langle\vec{\tau}'\rangle : \langle\Theta_m\lfloor^{\vec{\tau}}/_{\Theta^!_m}\rfloor\rangle \qquad \Psi \mid \Theta^! \mid \Sigma^\tau \vdash^t \Sigma^\tau_m\lfloor^{\vec{\tau},\vec{\tau}'}/_{\Theta^!_m,\Theta_m}\rfloor\end{array}}{\Psi \mid \Theta^! \mid \Sigma^\tau \vdash^t \tau^m.m\langle\vec{\tau}\rangle(\Gamma_m\lfloor^{\vec{\tau},\vec{\tau}'}/_{\Theta^!_m,\Theta_m}\rfloor) : \tau\lfloor^{\vec{\tau},\vec{\tau}'}/_{\Theta^!_m,\Theta_m}\rfloor}$$

$$\frac{\begin{array}{c}\Psi \mid \Theta^! \vdash \tau_m <: \tau'_m \qquad \Psi \mid \Theta^! \vdash (\Gamma) <: (\Gamma') \\ \Psi \mid \Theta^! \vdash \tau <: \tau' \qquad \Psi \mid \Theta^! \mid \Sigma^\tau \vdash \tau_m.m\langle\vec{\tau}\rangle(\Gamma') : \tau\end{array}}{\Psi \mid \Theta^! \mid \Sigma^\tau \vdash \tau'_m.m\langle\vec{\tau}\rangle(\Gamma) : \tau'}$$

Shape-Premise Term Validity $\Psi \mid \Theta^! \vdash^t [\Sigma^\tau]$

$$\frac{\Psi \mid \Theta^! \vdash_? \tau_1 \qquad ... \qquad \Psi \mid \Theta^! \vdash_? \sigma_1 \qquad ...}{\Psi \mid \Theta^! \vdash [\varsigma : \tau_1.\sigma_1, ...]}$$

Term Satisfaction $\Psi \mid \Theta^! \mid \Sigma^\tau \vdash^t \Sigma^\tau \quad \Psi \mid \Theta^! \mid \Sigma^\tau \vdash^t \tau.\sigma$

$$\frac{\Psi \mid \Theta^! \mid \Sigma^\tau \vdash^t \tau_1.\sigma_1 \qquad ...}{\Psi \mid \Theta^! \mid \Sigma^\tau \vdash^t \varsigma_1 : \tau_1.\sigma_1, ...} \qquad \frac{\begin{array}{c}\Psi \mid \Theta^! \mid \Sigma^\tau \vdash^t \tau.\sigma \\ \Psi \mid \Theta^! \vdash \tau' <: \tau \qquad \Psi \mid \Theta^! \vdash \sigma \sqsubset: \sigma'\end{array}}{\Psi \mid \Theta^! \mid \Sigma^\tau \vdash^t \tau'.\sigma'}$$

$$\frac{\varsigma : \tau.\sigma \in \Sigma^\tau}{\Psi \mid \Theta^! \mid \Sigma^\tau \vdash^t \tau.\sigma} \qquad \frac{}{\Psi \mid \Theta^! \mid \Sigma^\tau \vdash^t \bot.\sigma} \qquad \frac{\begin{array}{c}\Psi \mid \Theta^! \mid \Sigma^\tau \vdash^t \tau_1.\sigma \\ \Psi \mid \Theta^! \mid \Sigma^\tau \vdash^t \tau_2.\sigma\end{array}}{\Psi \mid \Theta^! \mid \Sigma^\tau \vdash^t (\tau_1 \cup \tau_2).\sigma}$$

$$\frac{\begin{array}{c}\textbf{interface}\ I\langle\Theta_I\rangle\ \textbf{extends}\ \bullet\ \textbf{satisfies}\ ..., \sigma'[\Theta'][\Sigma'], ...\ \{\bullet\} \in \Psi \\ \Psi \mid \Theta^! \vdash \langle\vec{\tau}^a\rangle \rightsquigarrow \langle\vec{\tau}\rangle \mid \Theta^!_a \qquad \Psi \mid \Theta^!, \Theta^!_a \vdash_? \langle\vec{\tau}'\rangle : \langle\Theta'\lfloor^{\vec{\tau}}/_{\Theta_I}\rfloor\rangle \\ \Psi \mid \Theta^!, \Theta^!_a \mid \Sigma^\tau \vdash^t \Sigma'\lfloor^{\vec{\tau},\vec{\tau}'}/_{\Theta_I,\Theta'}\rfloor \qquad \Psi \mid \Theta^!, \Theta^!_a \vdash \sigma'\lfloor^{\vec{\tau},\vec{\tau}'}/_{\Theta_I,\Theta'}\rfloor \sqsubset: \sigma\end{array}}{\Psi \mid \Theta^! \mid \Sigma^\tau \vdash^t I\langle\vec{\tau}^a\rangle.\sigma}$$

$$\frac{\begin{array}{c}\textbf{shape}\ S\langle\Theta_S\rangle\ \textbf{extends}\ \bullet\ \{\bullet\} \in \Psi \qquad \Psi \mid \Theta^! \mid \Sigma^\tau \vdash^t \tau.S\langle\vec{\tau}_1\rangle \\ \Psi \mid \Theta^! \mid \Sigma^\tau \vdash^t \tau.S\langle\vec{\tau}_2\rangle \qquad \Psi \mid \Theta^! \vdash \langle\vec{\tau}_1\rangle \cap \langle\vec{\tau}_2\rangle <: \langle\vec{\tau}\rangle : \langle\Theta_S\rangle\end{array}}{\Psi \mid \Theta^! \mid \Sigma^\tau \vdash^t \tau.S\langle\vec{\tau}\rangle}$$

Figure D.3 (contd.): Term Typing (Generics)

Method-Premise Validity $\Psi \mid \Theta^! \mid \Sigma^\tau \vdash^t s^\tau$

$$\frac{\begin{array}{c}\Psi \mid \Theta^! \vdash \langle \Theta_e^!, \Theta_i \rangle \\ \vdash \langle \Theta_i \rangle \rightsquigarrow \langle \Theta_i^! \rangle \qquad \Psi \mid \Theta^!, \Theta_e^!, \Theta_i^! \vdash (\Gamma_m) \qquad \Psi \mid \Theta^!, \Theta_e^!, \Theta_i^! \vdash^t [\Sigma_m^\tau] \\ \Psi \mid \Theta^!, \Theta_e^!, \Theta_i^! \vdash_+ \tau_m \qquad \Psi \mid \Theta^!, \Theta_e^! \vdash (\Gamma_m)[\Sigma_m] \rightsquigarrow \langle \Theta_i \mid \varnothing \rangle\end{array}}{\Psi \mid \Theta^! \mid \Sigma^\tau \vdash^t m\langle \Theta_e^! \rangle [\Theta_i](\Gamma_m)[\Sigma_m^\tau] : \tau_m}$$

Term Inheritance $\Psi \mid \Theta^! \mid \Sigma^\tau \vdash^t \{s^\tau; ...\} \sqsubseteq \tau$

$$\frac{\texttt{interface}\ I\langle \Theta_I \rangle\ \texttt{extends}\ \bullet\ \texttt{satisfies}\ \bullet \in \{s_1'; ...\} \in \Psi \qquad \Psi \mid \Theta^! \mid \Sigma^\tau \vdash^t \{s_1^\tau; ...\} \sqsubseteq s_1' \lfloor \vec{\tau}/\Theta_I \rceil \qquad ...}{\Psi \mid \Theta^! \mid \Sigma^\tau \vdash^t \{s_1^\tau; ...\} \sqsubseteq I\langle \vec{\tau} \rangle}$$

Method-Premise Inheritance $\Psi \mid \Theta^! \mid \Sigma^\tau \vdash \{s^\tau; ...\} \sqsubseteq s^\tau$

$$\frac{\Psi \mid \Theta^!, \Theta_e^!, \Theta_i \mid \Sigma^\tau, \Sigma_m^\tau \vdash \bot}{\Psi \mid \Theta^! \mid \Sigma^\tau \vdash \{s_1; ...\} \sqsubseteq m\langle \Theta_e^! \rangle [\Theta_i](\Gamma)[\Sigma_m^\tau] : \tau}$$

$$\frac{\begin{array}{cc} m\langle \Theta_1^! \rangle [\Theta_1](\Gamma_1)[\Sigma_1^\tau] : \tau_1 \in \{s_1^\tau; ...\} & \Psi \mid \Theta^!, \Theta_2^!, \Theta_2 \mid \Sigma^\tau, \Sigma_2^\tau \vdash \langle \Theta' \rangle [\Sigma'] \\ \Theta_2^! = \bullet <: \nu_1 \alpha_1 <: \bullet, ... & \Psi \mid \Theta' \vdash_? \langle \alpha_1, ... \rangle : \langle \Theta_1^! \rangle \\ \Psi \mid \Theta' \vdash_? \langle \vec{\tau} \rangle : \langle \Theta_1 \lfloor {}^{\alpha_1,...}/_{\Theta_1^!} \rceil \rangle & \Psi \mid \Theta' \vdash (\Gamma_2) <: (\Gamma_1 \lfloor {}^{\alpha_1,...,\vec{\tau}}/_{\Theta_1^!,\Theta_1} \rceil) \\ \Psi \mid \Theta' \mid \Sigma' \vdash \Sigma_1^\tau \lfloor {}^{\alpha_1,...,\vec{\tau}}/_{\Theta_1^!,\Theta_1} \rceil & \Psi \mid \Theta' \vdash \tau_1 \lfloor {}^{\alpha_1,...,\vec{\tau}}/_{\Theta_1^!,\Theta_1} \rceil <: \tau_2 \end{array}}{\Psi \mid \Theta^! \mid \Sigma^\tau \vdash \{s_1^\tau; ...\} \sqsubseteq m\langle \Theta_2^! \rangle [\Theta_2](\Gamma_2)[\Sigma_2^\tau] : \tau_2}$$

Program-Term Typing $\Psi \vdash^t t$

$$\frac{\vdash \Psi \qquad \Psi \mid \varnothing \mid \varnothing \mid \varnothing \vdash^t t : \top}{\vdash^t \Psi; t}$$

Figure D.3 (contd.): Term Typing (Generics)

Type Equivalence $\Psi \vdash \tau \approx \tau \quad \Psi \vdash \tau^a \approx \tau^a$

$$\frac{}{\Psi \vdash \alpha \approx \alpha} \qquad \frac{\Psi \mid \varnothing \vdash_? \tau \qquad \Psi \mid \varnothing \vdash_? \tau' \qquad \Psi \mid \varnothing \vdash \tau <: \tau' \qquad \Psi \mid \varnothing \vdash \tau' <: \tau}{\Psi \vdash \tau \approx \tau'}$$

$$\frac{\Psi \vdash \tau_i \approx \tau_i' \qquad \Psi \vdash \tau_o \approx \tau_o'}{\Psi \vdash \textbf{in } \tau_i \textbf{ out } \tau_o \approx \textbf{in } \tau_i' \textbf{ out } \tau_o'} \qquad \frac{\Psi \vdash \langle \vec{\tau}_1^a \rangle \approx \langle \vec{\tau}_2^a \rangle}{\Psi \vdash I\langle \vec{\tau}_1^a \rangle \approx I\langle \vec{\tau}_2^a \rangle}$$

$$\frac{\Psi \vdash \tau_1 \approx \tau_1' \qquad \Psi \vdash \tau_2 \approx \tau_2'}{\Psi \vdash \tau_1 \cup \tau_2 \approx \tau_1' \cup \tau_2'} \qquad \frac{\Psi \vdash \tau_1 \approx \tau_1' \qquad \Psi \vdash \tau_2 \approx \tau_2'}{\Psi \vdash \tau_1 \cap \tau_2 \approx \tau_1' \cap \tau_2'}$$

Type-Arguments Equivalence $\Psi \vdash \langle \vec{\tau}^a \rangle \approx \langle \vec{\tau}^a \rangle$

$$\frac{\Psi \vdash \tau_1^a \approx \tau_1^{a\prime} \qquad \ldots}{\Psi \vdash \langle \tau_1^a, \ldots \rangle \approx \langle \tau_1^{a\prime}, \ldots \rangle}$$

Shape Equivalence $\Psi \vdash \sigma \approx \sigma$

$$\frac{\Psi \vdash \langle \vec{\tau} \rangle \approx \langle \vec{\tau}' \rangle}{\Psi \vdash S\langle \vec{\tau} \rangle \approx S\langle \vec{\tau}' \rangle}$$

Kind-Context Equivalence $\Psi \vdash \langle \Theta \rangle \approx \langle \Theta \rangle$

$$\frac{}{\Psi \vdash \langle\rangle \approx \langle\rangle} \qquad \frac{\Psi \vdash \langle \Theta \rangle \approx \langle \Theta' \rangle \qquad \Psi \vdash \tau_\ell \approx \tau_\ell' \qquad \Psi \vdash \tau_u \approx \tau_u'}{\Psi \vdash \langle \Theta, \tau_\ell <: \nu\alpha <: \tau_u \rangle \approx \langle \Theta', \tau_\ell' <: \nu\alpha <: \tau_u' \rangle}$$

Program-Context Equivalence $\Psi \vdash (\Gamma) \approx (\Gamma) \quad \Psi \vdash (p) \approx (p)$

$$\frac{\Psi \vdash (p_1) \approx (p_1') \qquad \ldots}{\Psi \vdash (p_1; \ldots) \approx (p_1'; \ldots)}$$

$$\frac{\Psi \vdash \tau \approx \tau'}{\Psi \vdash (x : \tau) \approx (x : \tau')} \qquad \frac{\Psi \vdash \tau_1 \approx \tau_1' \qquad \ldots \qquad \Psi \vdash \tau \approx \tau'}{\Psi \vdash (f(\tau_1, \ldots) : \tau) \approx (f(\tau_1', \ldots) : \tau')}$$

Figure D.4: Type Equivalence

Shape-Context Equivalence $\Psi \vdash [\Sigma^\tau] \approx [\Sigma^\tau]$

$$\frac{\Psi \vdash \tau_1 \approx \tau_1' \quad \ldots \quad \Psi \vdash \sigma_1 \approx \sigma_1' \quad \ldots}{\Psi \vdash [\varsigma : \tau_1.\sigma_1, \ldots] \approx [\varsigma : \tau_1'.\sigma_1', \ldots]}$$

Signature Equivalence $\Psi \vdash s^\tau \approx s^\tau$

$$\frac{\begin{array}{c}\Psi \vdash \langle \Theta_1^! \rangle \approx \langle \Theta_2^! \rangle \quad \Psi \vdash \langle \Theta_1 \rangle \approx \langle \Theta_2 \rangle \\ \Psi \vdash (\Gamma_1) \approx (\Gamma_2) \quad \Psi \vdash [\Sigma_1^\tau] \approx [\Sigma_2^\tau] \quad \Psi \vdash \tau_1 \approx \tau_2\end{array}}{\Psi \vdash m\langle \Theta_1^! \rangle [\Theta_1](\Gamma_1)[\Sigma_1^\tau] : \tau_1 \approx m\langle \Theta_2^! \rangle [\Theta_2](\Gamma_2)[\Sigma_2^\tau] : \tau_2}$$

Term Equivalence $\Psi \vdash t \approx t \quad \Psi \vdash t_m \approx t_m$

$$\frac{}{\Psi \vdash x \approx x} \qquad \frac{\Psi \vdash t_1 \approx t_1' \quad \ldots}{\Psi \vdash f(t_1, \ldots) \approx f(t_1', \ldots)} \qquad \frac{}{\Psi \vdash \texttt{throw} \approx \texttt{throw}}$$

$$\frac{\Psi \vdash t_1^g \approx t_2^g \quad \Psi \vdash t_t \approx t_t' \quad \Psi \vdash t_f \approx t_f'}{\Psi \vdash \texttt{if } t_1^g \texttt{ then } t_t \texttt{ else } t_f \approx \texttt{if } t_2^g \texttt{ then } t_t' \texttt{ else } t_f'}$$

$$\frac{\Psi \vdash \langle \vec{\tau} \rangle \approx \langle \vec{\tau}' \rangle \quad \Psi \vdash d_1^t \approx d_1^{t'} \quad \ldots}{\Psi \vdash \texttt{object } x : I\langle \vec{\tau} \rangle \ \{d_1^t; \ldots\} \approx \texttt{object } x : I\langle \vec{\tau}' \rangle \ \{d_1^{t'}; \ldots\}}$$

$$\frac{\Psi \vdash t_1^m \approx t_2^m \quad \Psi \vdash \langle \vec{\tau} \rangle \approx \langle \vec{\tau}' \rangle \quad \Psi \vdash a_1^t \approx a_1^{t'} \quad \ldots}{\Psi \vdash t_1^m.m\langle \vec{\tau} \rangle (a_1^t; \ldots) \approx t_2^m.m\langle \vec{\tau}' \rangle (a_1^{t'}; \ldots)}$$

$$\frac{\Psi \vdash t \approx t'}{\Psi \vdash t@_\varsigma \approx t'@_\varsigma}$$

$$\frac{\Psi \vdash t \approx t' \quad \Psi \vdash t_x \approx t_x'}{\Psi \vdash \texttt{capture } t \texttt{ as } x : I\langle \vec{\alpha}^! \rangle \texttt{ in } t_x \approx \texttt{capture } t' \texttt{ as } x : I\langle \vec{\alpha}^! \rangle \texttt{ in } t_x'}$$

$$\frac{\Psi \vdash t \approx t'}{\Psi \vdash \texttt{capture } x \texttt{ as } I\langle \vec{\alpha}^! \rangle \texttt{ in } t \approx \texttt{capture } x \texttt{ as } I\langle \vec{\alpha}^! \rangle \texttt{ in } t'}$$

$$\frac{\Psi \vdash v \approx v' \quad \Psi \vdash t \approx t'}{\Psi \vdash \texttt{capture } v \texttt{ as } I\langle \vec{\alpha}^! \rangle \texttt{ in } t \approx \texttt{capture } v' \texttt{ as } I\langle \vec{\alpha}^! \rangle \texttt{ in } t'}$$

Figure D.5: Term Equivalence

Guard Equivalence $\Psi \vdash t^g \approx t^g$

$$\frac{\Psi \vdash \tau \approx \tau' \qquad \Psi \vdash \sigma \approx \sigma'}{\Psi \vdash \tau \ \mathbf{satisfies}\ \sigma \ \mathbf{as}\ \varsigma \approx \tau' \ \mathbf{satisfies}\ \sigma' \ \mathbf{as}\ \varsigma}$$

$$\frac{\Psi \vdash \tau_\ell \approx \tau'_\ell \qquad \Psi \vdash \tau_u \approx \tau'_u}{\Psi \vdash \tau_\ell <: \tau_u \approx \tau'_\ell <: \tau'_u} \qquad \frac{\Psi \vdash t \approx t' \qquad \Psi \vdash \tau \approx \tau'}{\Psi \vdash t \ \mathbf{is}\ x : \tau \approx t' \ \mathbf{is}\ x : \tau'}$$

$$\frac{\Psi \vdash \tau \approx \tau'}{\Psi \vdash x \ \mathbf{is}\ \tau \approx x \ \mathbf{is}\ \tau'} \qquad \frac{\Psi \vdash v \approx v' \qquad \Psi \vdash \tau \approx \tau'}{\Psi \vdash v \ \mathbf{is}\ \tau \approx v' \ \mathbf{is}\ \tau'}$$

Definition Equivalence $\Psi \vdash d^t \approx d^t$

$$\frac{\Psi \vdash s_1^\tau \approx s_2^\tau \qquad \Psi \vdash t_1 \approx t_2}{\Psi \vdash s_1^\tau \mapsto t_1 \approx s_2^\tau \mapsto t_2}$$

Argument Equivalence $\Psi \vdash a^t \approx a^t$

$$\frac{\Psi \vdash t \approx t'}{\Psi \vdash x \mapsto t \approx x \mapsto t'} \qquad \frac{\Psi \vdash t \approx t'}{\Psi \vdash f(x_1, \ldots) \mapsto t \approx f(x_1, \ldots) \mapsto t'}$$

Figure D.5 (contd.): Term Equivalence (Generics)

BIBLIOGRAPHY

Acay, Coşku and Frank Pfenning (2017). "Intersections and Unions of Session Types." In: *Electronic Proceedings in Theoretical Computer Science* 242.ITRS, pp. 4–19. DOI: 10.4204/EPTCS.242.3 (cit. on p. 127).

Ahmed, Amal, Robert Bruce Findler, Jeremy G. Siek, and Philip Wadler (2011). "Blame for All." In: *POPL 2011*. Austin, Texas, USA: ACM, pp. 201–214. ISBN: 978-1-4503-0490-0. DOI: 10.1145/1926385.1926409 (cit. on pp. 139, 176).

Ahn, Wonsun, Jiho Choi, Thomas Shull, María J. Garzarán, and Josep Torrellas (2014). "Improving JavaScript Performance by Deconstructing the Type System." In: *PLDI 2014*. Edinburgh, United Kingdom: ACM, pp. 496–507. ISBN: 978-1-4503-2784-8. DOI: 10.1145/2594291.2594332 (cit. on p. 183).

Aiken, Alexander and Edward L. Wimmers (1993). "Type Inclusion Constraints and Type Inference." In: *FPCA 1993*. Copenhagen, Denmark: ACM, pp. 31–41. ISBN: 0-89791-595-X. DOI: 10.1145/165180.165188 (cit. on pp. 92, 128).

Allende, Esteban, Oscar Callaú, Johan Fabry, Éric Tanter, and Marcus Denker (2014). "Gradual Typing for Smalltalk." In: *Science of Computer Programming* 96. Special issue on Advances in Smalltalk based Systems, pp. 52–69. ISSN: 0167-6423. DOI: 10.1016/j.scico.2013.06.006 (cit. on pp. 5, 8, 144, 148, 183, 193).

Allende, Esteban, Johan Fabry, and Éric Tanter (2013). "Cast Insertion Strategies for Gradually-Typed Objects." In: *DLS 2013*. Indianapolis, Indiana, USA: ACM, pp. 27–36. ISBN: 978-1-4503-2433-5. DOI: 10.1145/2508168.2508171 (cit. on p. 183).

Amin, Nada and Ross Tate (2016). "Java and Scala's Type Systems are Unsound: The Existential Crisis of Null Pointers." In: *OOPSLA 2016*. Amsterdam, Netherlands: ACM, pp. 838–848. ISBN: 978-1-4503-4444-9. DOI: 10.1145/2983990.2984004 (cit. on pp. 152, 283).

Ancona, Davide and Andrea Corradi (2016). "Semantic Subtyping for Imperative Object-oriented Languages." In: *OOPSLA 2016*. Amsterdam, Netherlands: ACM, pp. 568–587. ISBN: 978-1-4503-4444-9. DOI: 10.1145/2983990.2983992 (cit. on pp. 20, 69, 92, 127, 128).

Anderson, Carolyn Jane, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker (2014). "NetKAT: Semantic Foundations for Networks." In: *POPL 2014*. San Diego, California, USA: ACM, pp. 113–126. ISBN: 978-1-4503-2544-8. DOI: 10.1145/2535838.2535862 (cit. on p. 127).

Anderson, Christopher and Sophia Drossopoulou (2003). "BabyJ: From Object Based to Class Based Programming via Types." In: *WOOD 2003*. Amsterdam, Netherlands: Elsevier Science Publishers B. V., pp. 53–81. DOI: 10.1016/S1571-0661(04)80802-8 (cit. on p. 150).

Baader, Franz, Wayne Snyder, Paliath Narendran, Manfred Schmidt-Schauss, and Klaus Schulz (2001). "Chapter 8 - Unification Theory." In: *Handbook of Automated Reasoning*. Handbook of Automated Reasoning. Amsterdam, Netherlands: North-Holland, pp. 445–533. ISBN: 978-0-444-50813-3. DOI: 10.1016/B978-044450813-3/50010-2 (cit. on p. 23).

Bakel, Steffen van, Mariangiola Dezani-Ciancaglini, Ugo de'Liguoro, and Yoko Motohama (2000). *The Minimal Relevant Logic and the Call-by-Value Lambda Calculus*. Tech. rep. TR-ARP-05-2000. Australian National University (cit. on p. 119).

Balsters, Herman and Maarten M. Fokkinga (1991). "Subtyping can have a simple semantics." In: *Theoretical Computer Science* 87.1, pp. 81–96. ISSN: 0304-3975. DOI: 10.1016/S0304-3975(06)80005-8 (cit. on p. 273).

Barbanera, Franco, Mariangiola Dezani-Ciancaglini, and Ugo de'Liguoro (1995). "Intersection and Union Types: Syntax and Semantics." In: *Information and Computation* 119.2, pp. 202–230. ISSN: 0890-5401. DOI: 10.1006/inco.1995.1086 (cit. on p. 120).

Bauman, Spenser, Carl Friedrich Bolz-Tereick, Jeremy Siek, and Sam Tobin-Hochstadt (2017). "Sound Gradual Typing: Only Mostly Dead." In: *PACMPL* 1.OOPSLA, 54:1–54:24. ISSN: 2475-1421. DOI: 10.1145/3133878 (cit. on pp. 6, 143).

Bezanson, Jeff, Alan Edelman, Stefan Karpinski, and Viral B. Shah (2017). "Julia: A Fresh Approach to Numerical Computing." In: *SIAM Review* 59.1, pp. 65–98. DOI: 10.1137/141000671 (cit. on pp. 69, 126).

Bierman, Gavin, Martín Abadi, and Mads Torgersen (2014). "Understanding TypeScript." In: *ECOOP 2014*. Berlin, Heidelberg, Germany: Springer Berlin Heidelberg, pp. 257–281. ISBN: 978-3-662-44202-9. DOI: 10.1007/978-3-662-44202-9_11 (cit. on p. 69).

Bierman, Gavin, Erik Meijer, and Mads Torgersen (2010). "Adding Dynamic Types to C#." In: *ECOOP 2010*. Maribor, Slovenia: Springer Berlin Heidelberg, pp. 76–100. ISBN: 978-3-642-14107-2. DOI: 10.1007/978-3-642-14107-2_5 (cit. on pp. 6, 144, 146–148, 193, 344).

Bonchi, Filippo and Damien Pous (2013). "Checking NFA Equivalence with Bisimulations Up to Congruence." In: *POPL 2013*. Rome, Italy: ACM, pp. 457–468. ISBN: 978-1-4503-1832-7. DOI: 10.1145/2429069.2429124 (cit. on p. 127).

Bracha, Gilad (2004). "Pluggable Type Systems." In *Workshop on Revival of Dynamic Languages* (OOPSLA 2004). URL: http://bracha.org/pluggableTypesPosition.pdf (cit. on p. 148).

Brandt, Michael and Fritz Henglein (1997). "Coinductive axiomatization of recursive type equality and subtyping." In: pp. 63–81. DOI: 10.1007/3-540-62688-3_29 (cit. on p. 127).

Bruce, Kim B., Jonathan Crabtree, and Gerald Kanapathy (1994). "An operational semantics for TOOPLE: A statically-typed object-oriented programming language." In: *MFPS 1994*. Berlin, Heidelberg, Germany: Springer Berlin Heidelberg, pp. 603–626. ISBN: 978-3-540-48419-6. DOI: 10.1007/3-540-58027-1_30 (cit. on p. 273).

Bruce, Kim B., Martin Odersky, and Philip Wadler (1998). "A Statically Safe Alternative to Virtual Types." In: *ECOOP 1998*. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 523–549. ISBN: 978-3-540-69064-1. DOI: 10.1007/BFb0054106 (cit. on pp. 32, 64).

Canning, Peter S., William R. Cook, Walter L. Hill, Walter G. Olthoff, and John C. Mitchell (1989). "F-Bounded Polymorphism for Object-Oriented Programming." In: *FPCA 1989*. Imperial College, London, United Kingdom: ACM, pp. 273–280. ISBN: 0-89791-328-0. DOI: 10.1145/99370.99392 (cit. on pp. 23, 24, 32, 270).

Castagna, Giuseppe and Zhiwu Xu (2011). "Set-Theoretic Foundation of Parametric Polymorphism and Subtyping." In: *ICFP 2011*. Tokyo,

Japan: ACM, pp. 94–106. ISBN: 978-1-4503-0865-6. DOI: 10.1145/2034773.2034788 (cit. on pp. 20, 69, 128).

Chambers, C., D. Ungar, and E. Lee (1989). "An Efficient Implementation of SELF a Dynamically-typed Object-oriented Language Based on Prototypes." In: *OOPSLA 1989*. New Orleans, Louisiana, USA: ACM, pp. 49–70. ISBN: 0-89791-333-7. DOI: 10.1145/74877.74884 (cit. on p. 241).

Chung, Benjamin, Paley Li, Francesco Zappa Nardelli, and Jan Vitek (2018). "KafKa: Gradual Typing for Objects." In: *ECOOP 2018*. Vol. 109. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 12:1–12:24. ISBN: 978-3-95977-079-8. DOI: 10.4230/LIPIcs.ECOOP.2018.12 (cit. on p. 13).

Cimini, Matteo and Jeremy G. Siek (2016). "The Gradualizer: A Methodology and Algorithm for Generating Gradual Type Systems." In: *POPL 2016*. St. Petersburg, FL, USA: ACM, pp. 443–455. ISBN: 978-1-4503-3549-2. DOI: 10.1145/2837614.2837632 (cit. on pp. 148, 161, 193, 332).

Cimini, Matteo and Jeremy G. Siek (2017). "Automatically Generating the Dynamic Semantics of Gradually Typed Languages." In: *POPL 2017*. Paris, France: ACM, pp. 789–803. ISBN: 978-1-4503-4660-3. DOI: 10.1145/3009837.3009863 (cit. on p. 332).

Clebsch, Sylvan, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil (2015). "Deny Capabilities for Safe, Fast Actors." In: *AGERE! 2015*. Pittsburgh, PA, USA: ACM, pp. 1–12. ISBN: 978-1-4503-3901-8. DOI: 10.1145/2824815.2824816 (cit. on p. 68).

Coppo, Mario and Mariangiola Dezani-Ciancaglini (1978). "A New Type Assignment for λ-Terms." In: *Archiv für Mathematische Logik und Grundlagenforschung* 19.1, pp. 139–156. ISSN: 1432-0665. DOI: 10.1007/BF02011875 (cit. on p. 118).

Coppo, Mario, Mariangiola Dezani-Ciancaglini, and Patrick Sallé (1979). "Functional Characterization of Some Semantic Equalities Inside λ-Calculus." In: *Automata, Languages and Programming*. LNCS 71, pp. 133–146. DOI: 10.1007/3-540-09510-1_11 (cit. on p. 118).

Curien, Pierre-Louis and Giorgio Ghelli (1992). "Coherence of Subsumption, Minimum Typing and Type-Checking in F$\leq$." In: *Mathematical Structures in Computer Science* 2.1, pp. 55–91. DOI: 10.1017/S0960129500001134 (cit. on p. 72).

Damas, Luís and Robin Milner (1982). "Principal Type-Schemes for Functional Programs." In: *POPL 1982*. Albuquerque, New Mexico: ACM, pp. 207–212. ISBN: 0-89791-065-6. DOI: 10.1145/582153.582176 (cit. on pp. 71, 273).

Dardha, Ornela, Daniele Gorla, and Daniele Varacca (2013). "Semantic Subtyping for Objects and Classes." In: *FMOODS/FORTE 2013*. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 66–82. ISBN: 978-3-642-38592-6. DOI: 10.1007/978-3-642-38592-6_6 (cit. on pp. 20, 128, 129).

Demers, Alan, Mark Weiser, Barry Hayes, Hans Boehm, Daniel Bobrow, and Scott Shenker (1990). "Combining Generational and Conservative Garbage Collection: Framework and Implementations." In: *POPL 1990*. San Francisco, California, USA: ACM, pp. 261–269. ISBN: 0-89791-343-4. DOI: 10.1145/96709.96735 (cit. on pp. 183, 240).

Deutsch, L. Peter and Allan M. Schiffman (1984). "Efficient Implementation of the Smalltalk-80 System." In: *POPL 1984*. Salt Lake City, Utah, USA: ACM, pp. 297–302. ISBN: 0-89791-125-3. DOI: 10.1145/800017.800542 (cit. on p. 183).

Dürig, Michael (2010). *Scala Type Level Encoding of the SKI Calculus*. URL: http://michid.wordpress.com/2010/01/29/scala-type-level-encoding-of-the-ski-calculus/ (cit. on p. 19).

ECMA (June 2019). *ECMAScript(R) 2019 Language Specification*. http://www.ecma-international.org/ecma-262/. Version 10.0 (cit. on p. 3).

ECMA TC39-TG2 (2017). *C# Language Specification, 5th edition*. ECMA (cit. on pp. 276, 308, 377).

Ernst, Erik (2001). "Family Polymorphism." In: *ECOOP 2001*. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 303–326. ISBN: 978-3-540-45337-6. DOI: 10.1007/3-540-45337-7_17 (cit. on p. 32).

Facebook (2014). *The Flow Static Type Checker Documentation*. URL: http://flow.org/ (cit. on pp. 4, 8, 68).

Facebook (2016). *The Hack Language Specification, Version 1.1*. URL: https://github.com/hhvm/hack-langspec (cit. on pp. 4, 8, 148).

Feltey, Daniel, Ben Greenman, Christophe Scholliers, Robert Bruce Findler, and Vincent St-Amour (2018). "Collapsible Contracts: Fixing a Pathology of Gradual Typing." In: *PACMPL* 2.OOPSLA, 133:1–133:27. ISSN: 2475-1421. DOI: 10.1145/3276503 (cit. on pp. 6, 143, 256).

Findler, Robert Bruce and Matthias Felleisen (2002). "Contracts for Higher-Order Functions." In: *ICFP 2002*. Pittsburgh, PA, USA: ACM, pp. 48–59. ISBN: 1-58113-487-8. DOI: 10.1145/581478.581484 (cit. on p. 169).

Frisch, Alain, Giuseppe Castagna, and Véronique Benzaken (July 2002). "Semantic Subtyping." In: *LICS 2002*. Washington, DC, USA: IEEE, pp. 137–146. DOI: 10.1109/LICS.2002.1029823 (cit. on pp. 20, 68, 69, 128).

Frisch, Alain, Giuseppe Castagna, and Véronique Benzaken (2008). "Semantic Subtyping: Dealing Set-Theoretically with Function, Union, Intersection, and Negation Types." In: *Journal of the ACM* 55.4, 19:1–19:64. ISSN: 0004-5411. DOI: 10.1145/1391289.1391293 (cit. on pp. 20, 69, 92, 128).

Garcia, Ronald, Alison M. Clark, and Éric Tanter (2016). "Abstracting Gradual Typing." In: *POPL 2016*. St. Petersburg, FL, USA: ACM, pp. 429–442. ISBN: 978-1-4503-3549-2. DOI: 10.1145/2837614.2837670 (cit. on pp. 148, 153, 176, 178, 179, 193, 332, 344).

Gesbert, Nils, Pierre Genevès, and Nabil Layaïda (2015). "A Logical Approach to Deciding Semantic Subtyping." In: *TOPLAS* 38.1. DOI: 10.1145/2812805 (cit. on p. 127).

Ghelli, Giorgio and Benjamin C. Pierce (1998). "Bounded Existentials and Minimal Typing." In: *Theoretical Computer Science* 193.1, pp. 75–96. ISSN: 0304-3975. DOI: 10.1016/S0304-3975(96)00300-3 (cit. on p. 273).

Gochet, Paul, Pascal Gribomont, and Didier Rossetto (2005). "Algorithms for Relevant Logic." In: *Logic, Thought and Action*. Dordrecht, Netherlands: Springer Netherlands, pp. 479–496. ISBN: 978-1-4020-3167-0. DOI: 10.1007/1-4020-3167-X_21 (cit. on pp. 20, 68, 119).

Gosling, James, Bill Joy, Guy Steele, and Gilad Bracha (2005). *The Java™ Language Specification, 3rd Edition*. Boston, MA, USA: Addison-Wesley Professional. ISBN: 978-0-321-24678-3. URL: https://docs.oracle.com/javase/specs/jls/se6/html/j3TOC.html (cit. on pp. 30, 38, 67).

Gosling, James, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley (2015). *The Java® Language Specification, Java SE 8th Edition*. Boston, MA, USA: Addison-Wesley Professional. ISBN: 978-0-13-390069-9. URL: https:

//docs.oracle.com/javase/specs/jls/se8/html/index.html (cit. on pp. 308, 377).

Greenman, Ben and Matthias Felleisen (2018). "A Spectrum of Type Soundness and Performance." In: *PACMPL* 2.ICFP, 71:1–71:32. ISSN: 2475-1421. DOI: 10.1145/3236766 (cit. on pp. 6, 13, 143, 208).

Greenman, Ben and Zeina Migeed (2018). "On the Cost of Type-tag Soundness." In: *PEPM 2018*. Los Angeles, CA, USA: ACM, pp. 30–39. ISBN: 978-1-4503-5587-2. DOI: 10.1145/3162066 (cit. on pp. 6, 13).

Greenman, Ben, Fabian Muehlboeck, and Ross Tate (2014). "Getting F-Bounded Polymorphism into Shape." In: *PLDI 2014*. Edinburgh, United Kingdom: ACM, pp. 89–99. ISBN: 978-1-4503-2784-8. DOI: 10.1145/2594291.2594308 (cit. on pp. 15, 23).

Grigore, Radu (2017). "Java Generics are Turing Complete." In: *POPL 2017*. Paris, France: ACM, pp. 73–85. ISBN: 978-1-4503-4660-3. DOI: 10.1145/3009837.3009871 (cit. on pp. 12, 19, 271).

Gronski, Jessica, Kenneth Knowles, Aaron Tomb, Stephen N. Freund, and Cormac Flanagan (2006). "SAGE: Hybrid Checking for Flexible Specifications." In: *Scheme and Functional Programming Workshop 2016*. URL: http://schemeworkshop.org/2006/06-freund.pdf (cit. on pp. 1, 139).

Guha, Arjun, Claudiu Saftoiu, and Shriram Krishnamurthi (2010). "The Essence of JavaScript." In: *ECOOP 2010*. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 126–150. ISBN: 978-3-642-14107-2. DOI: 10.1007/978-3-642-14107-2_7 (cit. on p. 3).

Hejlsberg, Anders, Mads Torgersen, Scott Wiltamuth, and Peter Golde (2010). *The C# Programming Language, 4th Edition*. Addison-Wesley Professional. ISBN: 978-0-321-74176-9 (cit. on pp. 25, 27, 30).

Hejlsberg, Anders, Scott Wiltamuth, and Peter Golde (2005). *C# Language Specification 2.0*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc. (cit. on p. 127).

Henglein, Fritz (1994). "Dynamic Typing: Syntax and Proof Theory." In: *Science of Computer Programming* 22.3, pp. 197–230. ISSN: 0167-6423. DOI: 10.1016/0167-6423(94)00004-2 (cit. on p. 161).

Henglein, Fritz and Lasse Nielsen (2011). "Regular Expression Containment: Coinductive Axiomatization and Computational Interpretation." In: *POPL 2011*. Austin, Texas, USA: ACM, pp. 385–398. ISBN: 978-1-4503-0490-0. DOI: 10.1145/1926385.1926429 (cit. on p. 127).

Hindley, J. R. (1969). "The principal type scheme of an object in combinatory logic." In: *Transactions of the American Mathematical Society* 1.146, pp. 29–60. ISSN: 00029947. DOI: 10.2307/1995158 (cit. on pp. 273, 307).

Hosoya, Haruo and Benjamin C. Pierce (2003). "XDuce: A Statically Typed XML Processing Language." In: *ACM Transactions on Internet Technology* 3.2, pp. 117–148. DOI: 10.1145/767193.767195 (cit. on pp. 20, 69, 128).

Hosoya, Haruo, Jérôme Vouillon, and Benjamin C. Pierce (2000). "Regular Expression Types for XML." In: *ICFP 2000*. New York, NY, USA: ACM, pp. 11–22. ISBN: 1-58113-202-6. DOI: 10.1145/351240.351242 (cit. on p. 127).

Huang, Shan Shan, David Zook, and Yannis Smaragdakis (2007). "cJ: Enhancing Java with Safe Type Conditions." In: *AOSD 2007*. Vancouver, British Columbia, Canada: ACM, pp. 185–198. ISBN: 1-59593-615-7. DOI: 10.1145/1218563.1218584 (cit. on pp. 61, 65).

Igarashi, Atsushi and Benjamin C. Pierce (1999). "Foundations for Virtual Types." In: *ECOOP 1999*. Berlin, Heidelberg: Springer Berlin Heidelberg,

pp. 161–185. ISBN: 978-3-540-48743-2. DOI: 10.1007/3-540-48743-3_8 (cit. on p. 64).

Igarashi, Atsushi, Benjamin C. Pierce, and Philip Wadler (May 2001). "Featherweight Java: A Minimal Core Calculus for Java and GJ." In: *TOPLAS* 23.3, pp. 396–450. ISSN: 0164-0925. DOI: 10.1145/503502.503505 (cit. on pp. 151, 271).

Ihaka, Ross and Robert Gentleman (1996). "R: A Language for Data Analysis and Graphics." In: *Journal of Computational and Graphical Statistics* 5.3, pp. 299–314. DOI: 10.1080/10618600.1996.10474713 (cit. on p. 3).

Ina, Lintaro and Atsushi Igarashi (2011). "Gradual Typing for Generics." In: *OOPSLA 2011*. Portland, Oregon, USA: ACM, pp. 609–624. ISBN: 978-1-4503-0940-0. DOI: 10.1145/2048066.2048114 (cit. on pp. 148, 193, 199, 261, 271).

Jeannin, Jean-Baptiste, Dexter Kozen, and Alexandra Silva (2017). "CoCaml: Functional Programming with Regular Coinductive Types." In: *Fundamenta Informaticae* 150.3–4, pp. 347–377. DOI: 10.3233/FI-2017-1473 (cit. on p. 127).

JetBrains (2019). *Kotlin Language Documentation, Version 1.3*. URL: https://kotlinlang.org/docs/reference/ (cit. on pp. 19, 308, 377).

Kennedy, Andrew J. and Benjamin C. Pierce (2007). "On Decidability of Nominal Subtyping with Variance." In: *FOOL 2007*. New York, NY, USA: ACM (cit. on pp. 19, 25, 26, 28, 41, 46, 64, 127, 271).

King, Gavin (Nov. 2013). *The Ceylon Language Specification, Version 1.0* (cit. on pp. 19, 39, 70, 107, 111, 308, 377).

Komondoor, Raghavan, G. Ramalingam, Satish Chandra, and John Field (2005). "Dependent Types for Program Understanding." In: *TACAS*

2005. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 157–173. ISBN: 978-3-540-31980-1. DOI: 10.1007/978-3-540-31980-1_11 (cit. on p. 318).

Kozen, Dexter, Jens Palsberg, and Michael I. Schwartzbach (1995). "Efficient Recursive Subtyping." In: *Mathematical Structures in Computer Science* 5.1, pp. 113–125. DOI: 10.1017/S0960129500000657 (cit. on p. 127).

Kristensen, Bent Bruun, Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard (1983). "Abstraction Mechanisms in the BETA Programming Language." In: *POPL 1983*. Austin, Texas: ACM, pp. 285–298. ISBN: 0-89791-090-7. DOI: 10.1145/567067.567094 (cit. on p. 63).

Kuhlenschmidt, Andre, Deyaaeldeen Almahallawi, and Jeremy G. Siek (2019). "Toward Efficient Gradual Typing for Structural Types via Coercions." In: *PLDI*. Phoenix, AZ, USA: ACM, pp. 517–532. ISBN: 978-1-4503-6712-7. DOI: 10.1145/3314221.3314627 (cit. on pp. 6, 143, 241, 250).

Lattner, Chris and Vikram Adve (2004). "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation." In: *CGO 2004*. Palo Alto, California, pp. 75–86. DOI: 10.1109/CGO.2004.1281665 (cit. on p. 240).

Liquori, Luigi and Claude Stolze (2017). "A Decidable Subtyping Logic for Intersection and Union Types." In: *TTCS 2017*. Cham, CHE: Springer International Publishing, pp. 74–90. ISBN: 978-3-319-68953-1. DOI: 10.1007/978-3-319-68953-1_7 (cit. on p. 120).

Madsen, Ole Lehrmann and Birger Møller-Pedersen (1989). "Virtual Classes - A Powerful Mechanism in Object-Oriented Programming." In: *OOPSLA 1989*. New Orleans, Louisiana, USA: ACM, pp. 397–406. ISBN: 0-89791-333-7. DOI: 10.1145/74877.74919 (cit. on p. 63).

Matthews, Jacob and Robert Bruce Findler (2007). "Operational Semantics for Multi-Language Programs." In: *POPL 2007*. Nice, France: ACM, pp. 3–10. ISBN: 1-59593-575-4. DOI: 10.1145/1190216.1190220 (cit. on pp. 1, 138).

McBride, Conor (2002). "Faking it Simulating dependent types in Haskell." In: *Journal of Functional Programming* 12.4–5, pp. 375–392. DOI: 10.1017/S0956796802004355 (cit. on p. 19).

Microsoft (Oct. 2012). *TypeScript*. URL: http://www.typescriptlang.org/ (cit. on pp. 5, 8, 69, 148).

Microsoft (July 2018). *The Typescript Handbook*. URL: https://www.typescriptlang.org/docs/handbook/basic-types.html (cit. on p. 69).

Milner, Robin (1978). "A theory of type polymorphism in programming." In: *Journal of Computer and System Sciences* 17.3, pp. 348–375. ISSN: 0022-0000 (cit. on pp. 23, 307).

Mitchell, John C. (1988). "Polymorphic Type Inference and Containment." In: *Information and Computation* 76.2/3, pp. 211–249. DOI: 10.1016/0890-5401(88)90009-0 (cit. on p. 122).

Muehlboeck, Fabian and Ross Tate (2017a). "Sound Gradual Typing is Nominally Alive and Well." In: *OOPSLA 2017*. Vancouver, British Columbia, Canada: ACM, Article 56, Auxiliary Archive. DOI: 10.1145/3133880 (cit. on p. 145).

Muehlboeck, Fabian and Ross Tate (2017b). "Sound Gradual Typing is Nominally Alive and Well." In: *PACMPL* 1.OOPSLA, 56:1–56:30. ISSN: 2475-1421. DOI: 10.1145/3133880 (cit. on pp. 15, 145, 193).

Muehlboeck, Fabian and Ross Tate (2018a). "Empowering Union and Intersection Types with Integrated Subtyping." In: *OOPSLA 2018*. New

York, NY, USA: ACM, Article 112, Supplementary Material. DOI: 10.1145/3276482 (cit. on pp. 67, 90, 105, 107).

Muehlboeck, Fabian and Ross Tate (2018b). "Empowering Union and Intersection Types with Integrated Subtyping." In: *PACMPL* 2.OOPSLA, 112:1–112:29. ISSN: 2475-1421. DOI: 10.1145/3276482 (cit. on pp. 15, 67).

New, Max S. and Amal Ahmed (2018). "Graduality from Embedding-projection Pairs." In: *PACMPL* 2.ICFP, 73:1–73:30. ISSN: 2475-1421. DOI: 10.1145/3236768 (cit. on p. 332).

New, Max S., Daniel R. Licata, and Amal Ahmed (2019). "Gradual Type Theory." In: *PACMPL* 3.POPL, 15:1–15:31. ISSN: 2475-1421. DOI: 10.1145/3290328 (cit. on p. 332).

Odersky, Martin, Philippe Altherr, et al. (2014). *Scala Language Specification, Version 2.11*. URL: https://scala-lang.org/files/archive/spec/2.11/ (cit. on pp. 25, 30, 277, 301).

Odersky, Martin and Konstantin Läufer (1996). In: *POPL 1996*. St. Petersburg Beach, Florida, USA: ACM, pp. 54–67. ISBN: 0-89791-769-3. DOI: 10.1145/237721.237729 (cit. on p. 122).

Odersky, Martin and Matthias Zenger (2005). "Scalable Component Abstractions." In: *OOPSLA 2005*. San Diego, CA, USA: ACM, pp. 41–57. ISBN: 1-59593-031-0. DOI: 10.1145/1094811.1094815 (cit. on p. 64).

Oracle Corporation (2014). *Compatibility Guide for JDK 8*. URL: https://www.oracle.com/technetwork/java/javase/8-compatibility-guide-2156366.html (cit. on p. 268).

Palsberg, Jens and Christina Pavlopoulou (1998). "From Polyvariant Flow Information to Intersection and Union Types." In: *POPL 1998*. San Diego, California, USA: ACM, pp. 197–208. ISBN: 0-89791-979-3. DOI: 10.1145/268946.268963 (cit. on p. 67).

Peyton Jones, Simon L., Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields (2007). "Practical Type Inference for Arbitrary-Rank Types." In: *Journal of Functional Programming* 17.1, pp. 1–82. DOI: 10.1017/S0956796806006034 (cit. on pp. 92, 121).

Pierce, Benjamin C. (1989). *A Decision Procedure for the Subtype Relation on Intersection Types with Bounded Variables*. Technical Report CMU-CS-89-169. Carnegie Mellon University (cit. on pp. 90, 91, 106).

Pierce, Benjamin C. (Feb. 1991). *Programming with Intersection Types, Union Types, and Polymorphism*. Technical Report CMU-CS-91-106. Carnegie Mellon University (cit. on pp. 68, 92, 119).

Pierce, Benjamin C. (1992). "Bounded Quantification is Undecidable." In: *POPL 1992*. Albuquerque, New Mexico, USA: ACM, pp. 305–315. ISBN: 0-89791-453-8. DOI: 10.1145/143165.143228 (cit. on pp. 19, 307).

Pierce, Benjamin C. (2002). *Types and Programming Languages*. Boston, MA, USA: MIT Press (cit. on p. 81).

Pierce, Benjamin C. and Martin Steffen (1997). "Higher-Order Subtyping." In: *Theoretical Computer Science* 176.1-2, pp. 235–282. ISSN: 0304-3975. DOI: 10.1016/S0304-3975(96)00096-5 (cit. on pp. 29, 56).

Politz, Joe Gibbs, Alejandro Martinez, Matthew Milano, Sumner Warren, Daniel Patterson, Junsong Li, Anand Chitipothu, and Shriram Krishnamurthi (2013). "Python: The Full Monty." In: *OOPSLA 2013*. Indianapolis, Indiana, USA: ACM, pp. 217–232. ISBN: 978-1-4503-2374-1. DOI: 10.1145/2509136.2509536 (cit. on p. 3).

Pottinger, Garrell (1980). "A Type Assignment for the Strongly Normalizable λ-Terms." In: *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*. London, GBR, New York, NY, USA: Academic Press (cit. on p. 119).

Rastogi, Aseem, Nikhil Swamy, Cédric Fournet, Gavin Bierman, and Panagiotis Vekris (2015). "Safe & Efficient Gradual Typing for TypeScript." In: *POPL 2015*. Mumbai, India: ACM, pp. 167–180. ISBN: 978-1-4503-3300-9. DOI: 10.1145/2676726.2676971 (cit. on pp. 136, 140, 144, 148).

Reynolds, John C. (Jan. 1988). *Preliminary Design of the Programming Language Forsythe*. Technical Report CMU-CS-88-159. Carnegie Mellon University (cit. on pp. 68, 90, 92, 122, 284).

Reynolds, John C. (1997). "Design of the Programming Language Forsythe." In: *ALGOL-like Languages*. Boston, MA, USA: Birkhäuser Boston, pp. 173–233. DOI: 10.1007/978-1-4612-4118-8_9 (cit. on pp. 68, 90, 122, 284).

Richards, Gregor, Ellen Arteca, and Alexi Turcotte (2017). "The VM Already Knew That: Leveraging Compile-time Knowledge to Optimize Gradual Typing." In: *PACMPL* 1.OOPSLA, 55:1–55:27. ISSN: 2475-1421. DOI: 10.1145/3133879 (cit. on pp. 6, 143).

Richards, Gregor, Francesco Zappa Nardelli, and Jan Vitek (2015). "Concrete Types for TypeScript." In: *ECOOP 2015*. Vol. 37. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, pp. 76–100. ISBN: 978-3-939897-86-6. DOI: 10.4230/LIPIcs.ECOOP.2015.76 (cit. on pp. 144, 148, 150, 193).

Rompf, Tiark and Nada Amin (2016). "Type Soundness for Dependent Object Types (DOT)." In: *OOPSLA 2016*. Amsterdam, Netherlands: ACM, pp. 624–641. ISBN: 978-1-4503-4444-9. DOI: 10.1145/2983990.2984008 (cit. on p. 68).

Rossum, Guido van (May 1995). *Python Tutorial*. Tech. rep. CS-R9526. Centrum for Wiskunde en Informatica (cit. on p. 3).

Routley, Richard and Robert K. Meyer (1972). "The Semantics of Entailment: III." In: *Journal of Philosophical Logic* 1.2, pp. 192–208. DOI: 10.2307/30226036 (cit. on pp. 68, 119).

Schinz, Michel (2005). "Compiling Scala for the Java Virtual Machine." PhD thesis. EPFL (cit. on p. 261).

Siek, Jeremy G., Ronald Garcia, and Walid Taha (2009). "Exploring the Design Space of Higher-Order Casts." In: *ESOP 2009*. York, UK: Springer Berlin Heidelberg, pp. 17–31. ISBN: 978-3-642-00589-3. DOI: 10.1007/978-3-642-00590-9_2 (cit. on p. 176).

Siek, Jeremy G. and Walid Taha (2006). "Gradual Typing for Functional Languages." In: *Scheme and Functional Programming Workshop* 6, pp. 81–92 (cit. on pp. 1, 137, 138, 161, 169, 193).

Siek, Jeremy G. and Walid Taha (2007). "Gradual Typing for Objects." In: *ECOOP 2007*. Berlin, Germany: Springer Berlin Heidelberg, pp. 2–27. ISBN: 978-3-540-73589-2. DOI: 10.1007/978-3-540-73589-2_2 (cit. on pp. 143, 155, 161, 193).

Siek, Jeremy G., Michael M. Vitousek, Matteo Cimini, and John Tang Boyland (2015). "Refined Criteria for Gradual Typing." In: *SNAPL 2015*. Vol. 32. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, pp. 274–293. ISBN: 978-3-939897-80-4. DOI: 10.4230/LIPIcs.SNAPL.2015.274 (cit. on pp. 6, 12, 141, 142, 178, 179, 181, 200, 219, 332).

Siek, Jeremy G., Michael M. Vitousek, Matteo Cimini, Sam Tobin-Hochstadt, and Ronald Garcia (2015). "Monotonic References for Efficient Gradual Typing." In: *ESOP 2015*. London, UK: Springer Berlin Heidelberg, pp. 432–456. ISBN: 978-3-662-46669-8. DOI: 10.1007/978-3-662-46669-8_18 (cit. on pp. 13, 136, 140, 233).

Siek, Jeremy G. and Philip Wadler (2010). "Threesomes, with and without Blame." In: *POPL 2010*. Madrid, Spain: ACM, pp. 365–376. ISBN: 978-1-60558-479-9. DOI: 10.1145/1706299.1706342 (cit. on pp. 175, 177, 256).

Smith, Daniel and Robert Cartwright (2008). "Java Type Inference is Broken: Can We Fix It?" In: *OOPSLA 2008*. Nashville, TN, USA: ACM, pp. 505–524. ISBN: 978-1-60558-215-3. DOI: 10.1145/1449764.1449804 (cit. on pp. 14, 29, 51, 65, 152).

Swamy, Nikhil, Cedric Fournet, Aseem Rastogi, Karthikeyan Bhargavan, Juan Chen, Pierre-Yves Strub, and Gavin Bierman (2014). "Gradual Typing Embedded Securely in JavaScript." In: *POPL 2014*. San Diego, California, USA: ACM, pp. 425–437. ISBN: 978-1-4503-2544-8. DOI: 10.1145/2535838.2535889 (cit. on pp. 5, 8, 136, 140, 142, 144, 147, 148, 193).

Takikawa, Asumu, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek, and Matthias Felleisen (2016). "Is Sound Gradual Typing Dead?" In: *POPL 2016*. St. Petersburg, FL, USA: ACM, pp. 456–468. ISBN: 978-1-4503-3549-2. DOI: 10.1145/2837614.2837630 (cit. on pp. 6, 136, 140, 142, 146, 148, 150, 181, 182, 184, 185, 239, 247, 248).

Tate, Ross (2013). "Mixed-Site Variance." In: *FOOL* (cit. on pp. 30, 43, 283).

Tate, Ross, Alan Leung, and Sorin Lerner (2011). "Taming Wildcards in Java's Type System." In: *PLDI 2011*. San Jose, California, USA: ACM, pp. 614–627. ISBN: 978-1-4503-0663-8. DOI: 10.1145/1993498.1993570 (cit. on pp. 27–30, 41, 46, 47, 51, 64, 65, 116).

Tempero, Ewan, Craig Anslow, Jens Dietrich, Ted Han, Jing Li, Markus Lumpe, Hayden Melton, and James Noble (2010). "Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies." In: *APSEC 2010*, pp. 336–345. DOI: 10.1109/APSEC.2010.46 (cit. on p. 36).

The Coq Development Team (1984). *The Coq Proof Assistant*. URL: https://coq.inria.fr/ (cit. on p. 70).

The Dotty Development Team (2015). *Dotty*. URL: http://dotty.epfl.ch/ (cit. on p. 68).

The LLVM Project (Aug. 2019). *LLVM*. URL: https://www.llvm.org (cit. on p. 240).

The MathWorks, Inc. (Aug. 2019). *MATLAB*. URL: https://www.mathworks.com/products/matlab.html (cit. on p. 3).

The PHP Group (2019). *PHP Language Reference*. https://www.php.net/manual/en/langref.php. Version 7.4.0 (cit. on p. 3).

The Python Development Team (Dec. 2008). *Python Benchmarks*. https://hg.python.org/benchmarks/ (cit. on p. 182).

Thorup, Kresten Krab (1997). "Genericity in Java with Virtual Types." In: *ECOOP*. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 444–471. ISBN: 978-3-540-69127-3. DOI: 10.1007/BFb0053390 (cit. on p. 64).

Thorup, Kresten Krab and Mads Torgersen (1999). "Unifying Genericity - Combining the Benefits of Virtual Types and Parameterized Classes." In: *ECOOP*. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 186–204. ISBN: 978-3-540-48743-2. DOI: 10.1007/3-540-48743-3_9 (cit. on p. 64).

Tobin-Hochstadt, Sam and Matthias Felleisen (2006). "Interlanguage Migration: From Scripts to Programs." In: *OOPSLA 2006*. Portland, Oregon, USA: ACM, pp. 964–974. ISBN: 1-59593-491-X. DOI: 10.1145/1176617.1176755 (cit. on pp. 136, 138, 141, 169, 176).

Tobin-Hochstadt, Sam and Matthias Felleisen (2008). "The Design and Implementation of Typed Scheme." In: *POPL 2008*. San Francisco, Cali-

fornia, USA: ACM, pp. 395–406. ISBN: 978-1-59593-689-9. DOI: 10.1145/1328438.1328486 (cit. on pp. 1, 5, 8, 13, 68, 193, 207, 250, 318).

Torgersen, Mads (1998). "Virtual Types are Statically Safe." In: *FOOL 1998* (cit. on p. 64).

Torgersen, Mads, Christian Plesner Hansen, Erik Ernst, Peter von der Ahé, Gilad Bracha, and Neal Gafter (2004). "Adding Wildcards to the Java Programming Language." In: *SAC 2004*. Nicosia, Cyprus: ACM, pp. 1289–1296. ISBN: 1-58113-812-1. DOI: 10.1145/967900.968162 (cit. on pp. 27, 51, 64).

Toro, Matías, Elizabeth Labrada, and Éric Tanter (2019). "Gradual Parametricity, Revisited." In: *PACMPL* 3.POPL, 17:1–17:30. ISSN: 2475-1421. DOI: 10.1145/3290330 (cit. on p. 332).

Toro, Matías and Éric Tanter (2017). "A Gradual Interpretation of Union Types." In: *SAS 2017*. Cham: Springer International Publishing, pp. 382–404. ISBN: 978-3-319-66706-5. DOI: 10.1007/978-3-319-66706-5_19 (cit. on pp. 176, 177).

Verlaguet, Julien and Alok Menghrajani (Mar. 2014). *Hack: a new programming language for HHVM*. https://engineering.fb.com/developer-tools/hack-a-new-programming-language-for-hhvm/ (cit. on p. 4).

Viganò, Luca (2000). "An O(n log n)-Space Decision Procedure for the Relevance Logic B+." In: *Studia Logica* 66.3, pp. 385–407. ISSN: 1572-8730. DOI: 10.1023/A:1005212701420 (cit. on pp. 21, 68, 119).

Viroli, Mirko (Sept. 2000). *On the Recursive Generation of Parametric Types*. Tech. rep. DEIS-LIA-00-002. University of Bologna (cit. on pp. 46, 127).

Vitousek, Michael M., Andrew M. Kent, Jeremy G. Siek, and Jim Baker (2014). "Design and Evaluation of Gradual Typing for Python." In: *DLS*

2014. Portland, Oregon, USA: ACM, pp. 45–56. ISBN: 978-1-4503-3211-8. DOI: 10.1145/2661088.2661101 (cit. on pp. 5, 8, 13, 136, 139, 140, 144, 146, 148, 193, 208, 233).

Vitousek, Michael M., Cameron Swords, and Jeremy G. Siek (2017). "Big Types in Little Runtime: Open-World Soundness and Collaborative Blame for Gradual Type Systems." In: *POPL 2017*. Paris, France: ACM, pp. 762–774. ISBN: 978-1-4503-4660-3. DOI: 10.1145/3009837.3009849 (cit. on pp. 6, 140, 143, 182, 187, 188, 193, 208, 233, 255).

Wadler, Philip and Stephen Blott (1989). "How to Make Ad-hoc Polymorphism Less Ad Hoc." In: *POPL 1989*. Austin, Texas, USA: ACM, pp. 60–76. ISBN: 0-89791-294-2. DOI: 10.1145/75277.75283 (cit. on pp. 61, 277).

Wadler, Philip and Robert Bruce Findler (2009). "Well-Typed Programs Can't Be Blamed." In: *ESOP 2009*. York, UK: Springer Berlin Heidelberg, pp. 1–16. ISBN: 978-3-642-00590-9. DOI: 10.1007/978-3-642-00590-9_1 (cit. on pp. 139, 141, 176, 178, 207).

Wehr, Stefan, Ralf Lämmel, and Peter Thiemann (2007). "JavaGI: Generalized Interfaces for Java." In: *ECOOP 2007*. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 347–372. ISBN: 978-3-540-73589-2. DOI: 10.1007/978-3-540-73589-2_17 (cit. on pp. 61, 65).

Wells, Joe B., Allyn Dimock, Robert Muller, and Franklyn Turbak (2002). "A Calculus with Polymorphic and Polyvariant Flow Types." In: *Journal of Functional Programming* 12.3, pp. 183–227. DOI: 10.1017/S0956796801004245 (cit. on p. 67).

"Revised Report on the Algorithmic Language ALGOL 68" (1975). In: *Acta Informatica* 5.1. Ed. by Adriaan van Wijngaarden, Barry J. Mailloux, John E. L. Peck, Cornelis H. A. Koster, Michel Sintzoff, Charles H. Lindsey,

Lambert G. L. T. Meertens, and Richard G. Fisker, pp. 1–236. ISSN: 1432-0525. DOI: 10.1007/BF00265077 (cit. on p. 119).

Wrigstad, Tobias, Francesco Zappa Nardelli, Sylvain Lebresne, Johan Östlund, and Jan Vitek (2010). "Integrating Typed and Untyped Code in a Scripting Language." In: *POPL 2010*. Madrid, Spain: ACM, pp. 377–388. ISBN: 978-1-60558-479-9. DOI: 10.1145/1706299.1706343 (cit. on pp. 150, 194, 239).

Zappa Nardelli, Francesco, Julia Belyakova, Artem Pelenitsyn, Benjamin Chung, Jeff Bezanson, and Jan Vitek (2018). "Julia Subtyping: A Rational Reconstruction." In: *PACMPL* 2.OOPSLA, 113:1–113:27. ISSN: 2475-1421. DOI: 10.1145/3276483 (cit. on pp. 19, 69, 126).

Zhang, Yizhou, Matthew C. Loring, Guido Salvaneschi, Barbara Liskov, and Andrew C. Myers (2015). "Lightweight, Flexible Object-oriented Generics." In: *PLDI 2015*. Portland, OR, USA: ACM, pp. 436–445. ISBN: 978-1-4503-3468-6. DOI: 10.1145/2737924.2738008 (cit. on pp. 64, 277, 301).

Zhang, Yizhou and Andrew C. Myers (2017). "Familia: Unifying Interfaces, Type Classes, and Family Polymorphism." In: *PACMPL* 1.OOPSLA, 70:1–70:31. ISSN: 2475-1421. DOI: 10.1145/3133894 (cit. on p. 64).