# Inferable Existential Quantification

Ross Tate[1], Juan Chen[2], and Chris Hawblitzel[2]

[1] University of California, San Diego
[2] Microsoft Research, Redmond

**Abstract.** Abstract interpretation is an excellent tool for designing complete dataflow analyses [6]. Existential quantification is an excellent tool for designing precise dataflow analyses. However, these two tools are difficult to combine. Abstract interpretation requires algorithms for deciding subtypes and constructing joins. This is problematic for existential quantification because simply subtyping with existential quantification is undecidable in general [17], not to mention the additional challenge of joining with existential quantification.
This paper presents a category-theoretic framework for designing abstract domains for complete inference and analysis using constrained but expressive forms of existential quantification. This framework is constructive in that it supplies the abstract algorithms for deciding subtypes and constructing joins. This framework is also instructive in that it provides guidelines to follow while designing the abstract domain and existential quantification in order to guarantee inferability. Finally, the framework is practical for real-world applications, as demonstrated by its critical role in designing an inferable typed assembly language for C# [16].

## 1 Introduction

Existential quantification grants a great deal of precision. In typed assembly languages [14], existential quantification over classes with inheritance constraints has been used to verify dynamic dispatch and runtime casts [3–5, 11, 16]; existential quantification over integers with ordering constraints has been used to verify array accesses [12, 16]; and existential quantification over arbitrary types has been used to verify closures [13, 14]. Existential quantification is particularly useful because it can track connections between separate abstract values. For example, existential quantification can express that an instance in one register and a vtable in another register both correspond to the *same* class which implements `Shape`:

$$\exists \alpha \ll \texttt{Shape}.\{\texttt{EAX} \mapsto \textsc{Ins}(\alpha), \texttt{EBX} \mapsto \textsc{VTable}(\alpha)\}$$

Or existential quantification can express that the integer in one register can be used to access the array in another register:

$$\exists i <_{32+} \ell.\{\texttt{EAX} \mapsto \textsc{Int}(4 * i), \texttt{EBX} \mapsto \textsc{Ins}(\texttt{Square})[\ell]\}$$

However, the expressiveness of existential quantification comes with a price. Subtyping with existential quantification is generally undecidable [17], so these typed assembly languages would often use **pack** and **open/unpack** pseudo-instructions, provided by the type-preserving compiler, essentially in order to inform the type checker how to subtype the existential types [4, 5, 11, 14]. Similarly, each merge point in the control-flow graph would be annotated with the existential type at that point so that the type checker would never need to compute joins of existential types [4, 5].

For our inferable typed assembly language [16], these annotations were unavailable and so we had to determine some way to infer types even with existential quantification. However, we still had at least function signatures available to us, so we decided to apply forward *abstract interpretation* [6]. This was no small challenge. As we would find out, it can be quite difficult to predict which features are or are not inferable in the presence of existential quantification using abstract interpretation. For example, generic classes, a more recent addition to C#, required almost no change to our type inference algorithm, whereas classic null pointers, which are typically straightforward to incorporate into inference, turn out to make type inference incomplete in the presence of existential quantification. Complete inference was important to us because we wanted our type checking algorithm to be predictable. The compiler writer should be able to know which program transformations will preserve typeability, and an executable which type checks with one valid implementation of the type checker should type check with all valid implementations of the type checker. Thus we would prefer a type system with a predictable inference algorithm over a more expressive type system but with an incomplete inference algorithm.

Recognizing early on that existential quantification would be the key challenge to designing an inferable typed assembly language, we created a category-theoretic framework for designing existentially quantified abstract domains

with a complete inference algorithm. This framework provided us with a lot of flexibility as we designed our typed assembly language, even enabling us to trade off between precision and efficiency while maintaining predictability. Our framework is instructive in that it provides guidelines to follow while designing the existentially quantified abstract domain. These guidelines were critical to designing our inferable typed assembly language. For example, they informed us that the standard null-pointer type would likely be problematic, and we went on to prove this to be the case and designed another way to handle null pointers. Our framework is also constructive in that it provides the abstract algorithms required by abstract interpretation. Our implementation was almost completely specified by these abstract algorithms [16].

Because our framework is expressed categorically, it can be applied to many abstract domains besides our original application. Not only can our framework be used to design an inference algorithm for a setting which already has existential quantification, but it can also be used to add existential quantification to an existing abstract domain in order to produce a more expressive abstract domain and precise analysis. In short, our framework brings existential quantification into the settings of complete type inference and complete program analysis.

In this paper we make the following contributions:

- In Section 2, we explain existential quantification and its categorical interpretation, separating an existentially quantified abstract domain into *bounds* and *bodies*.
- In Section 3, we present our restrictions on the *bodies* of an existentially quantified abstract domain, closing with our abstract algorithm for subtyping with existential quantification.
- In Section 4, we present our restrictions on the *bounds* of an existentially quantified abstract domain, closing with our abstract algorithm for joining with existential quantification.
- In Section 5, we present tools for proving co-well-foundedness and monotonicity with existential quantification as required by abstract interpretation.
- In Section 6, we present techniques for incorporating nested existential quantification into an existentially quantified abstract domain.

These components form a complete inference or analysis algorithm using abstract interpretation. Lastly in Section 7, we conclude the paper with a summary of our framework, connections to related work, and opportunities for future work. In our appendices, we provide a toolset for designing entirely new expressive forms of existential quantification meeting the requirements of our framework so that our framework can be applied to settings entirely different from those we have encountered and present here.

## 2 Existential Quantification

Before we get into existential quantification, first consider an abstract domain without any quantification whatsoever. Elements in such an abstract domain (which we will call *simple types* $\tau$) might look like the following:

$$\{\texttt{EAX} \mapsto \textsc{Ins}(\texttt{Square}), \texttt{EBX} \mapsto \textsc{Int}(5)\}$$

This simple type abstracts all concrete states $\sigma$ such that `EAX` contains a pointer to some instance of exactly `Square` and `EBX` contains the 32-bit representation of 5. Now we can add another layer of abstraction by using existential quantification:

$$\exists \alpha \ll \texttt{Shape}, i <_{32+} 5.\{\texttt{EAX} \mapsto \textsc{Ins}(\alpha), \texttt{EBX} \mapsto \textsc{Int}(3 * i + 2)\}$$

Here $\alpha$ represents an statically unknown class which is statically known to extend `Shape`, and $i$ represents a statically unknown integer whose unsigned 32-bit representation is statically known to be strictly less than that of 5. This existentially quantified abstract domain now abstracts the original abstract domain of simple types:

$$\frac{\vartheta \text{ is a valid mapping of variables in } \Gamma \text{ to constants} \qquad \tau \leq \tau'[\vartheta]}{\tau :: \exists \Gamma.\tau'}$$

We use double-lines to indicate that the above inference rule is 2-way (i.e. it is an if-and-only-if statement). The judgement $\tau :: \exists \Gamma.\tau'$ indicates that the simple type $\tau$ is abstracted by the existential type $\exists \Gamma.\tau'$. The judgement $\tau \leq \tau'[\vartheta]$ indicates that $\tau$ is smaller than $\tau'[\vartheta]$ in the *original* abstract domain. We can go one more step and compose the two layers of abstraction to see which concrete states are abstracted by an existential type:

$$\frac{\vartheta \text{ is a valid mapping of variables in } \Gamma \text{ to constants} \qquad \sigma :: \tau[\vartheta]}{\sigma :: \exists \Gamma.\tau}$$

Thus existential quantification simply serves as an additional layer of abstraction. However, we have yet to define the ordering on this existentially quantified abstract domain. This ordering is key to understanding and using existential quantification.

### 2.1 Existential Subtyping

Orderings on existential types (which we will call *existential subtyping*) are actually fairly simple:

$$\frac{\theta \text{ is a } \textit{permitted} \text{ mapping from } \Gamma' \text{ to } \Gamma \qquad \tau \leq \tau'[\theta]}{\exists \Gamma.\tau \sqsubseteq \exists \Gamma'.\tau'}$$

The mapping $\theta$ can be composed with any valid mapping $\vartheta$ from $\Gamma$ to constants to produce a valid mapping $\theta\,;\vartheta$ (i.e. $\vartheta \circ \theta$) from $\Gamma'$ to constants; thus any simple type or concrete state abstracted by $\exists\Gamma.\tau$ is also abstracted by $\exists\Gamma'.\tau'$. Note that we use $\leq$ for subtyping without quantification, and $\sqsubseteq$ for existential subtyping.

Now, we purposely used the word *permitted* above rather than *valid*, because choosing which mappings are permitted grants a lot of important flexibility to the designer of the existentially quantified domain. For example, consider the following two existential types (where $c$ is any constant between 1 and $2^{32}-2$):

$$\exists i <_{32+} c.\{\texttt{EAX} \mapsto \textsc{Int}(i+1)\} \text{ and } \exists j <_{32+} c+1.\{\texttt{EAX} \mapsto \textsc{Int}(j)\}$$

Mapping $j$ to $i+1$ would be *valid*; however, permitting this mapping for any $c$ would actually produce an abstract domain whose height is at least $2^{32}-2$. This abstract domain would be impractical in an analysis setting; it would take at least $2^{32}-2$ iterations to reach a fixed point for the following common program:

```
if (EAX < 1)
  while (EAX < arr.length)
    arr[EAX]++;
    EAX++;
```

Thus the designer may choose not to permit mappings like the one above in order to produce an efficient analysis. Also, the designer of the abstract domain may prefer to not have to include a decision procedure for constraint inclusion of inequalities in modular arithmetic within their analysis (or type checker in the case of a typed assembly language). Thus, the designer may choose to permit maps whose integer constraints can be satisfied by using only simple laws such as reflexivity and transitivity or other laws which they know are particularly useful for their application. Or they may decide to disallow expressions such as $i+1$ and only use integer variables. Regardless, our framework grants the flexibility for the designer to make these decisions depending on what they think is appropriate for the situation.

### 2.2 A Category of Bounds

Once the designer of the existential types has decided what kind of existential bounds they would like (e.g. classes with inheritance or integers expressions with inequalities) and what mappings they permit, they have essentially specified a *category* $\mathbf{Bnd}$ of bounds and mappings.

A category $\mathbf{C}$ has two components and two operations that must satisfy two properties. Categories are an abstract concept, so we interleave the definition of $\mathbf{Set}$, the category of sets and functions, in order to provide the reader with a concrete example to connect these abstract concepts to. Similarly, we show how the existential bounds and mappings specified by the designer form a category $\mathbf{Bnd}$. The two components of a category are:

- A set $\mathcal{O}bj$ of objects, like vertices in a multigraph
  In $\mathbf{Set}$, $\mathcal{O}bj$ is the set of sets (putting details aside).
  In $\mathbf{Bnd}$, each object is a bound $\Gamma$ which can be used in an existential type (as in $\exists\Gamma.\tau$).
- For each two objects $\mathcal{A}$ and $\mathcal{B}$, a set $\mathcal{M}or(\mathcal{A},\mathcal{B})$ of morphisms from $\mathcal{A}$ to $\mathcal{B}$, like directed edges in a multigraph
  In $\mathbf{Set}$, the set of morphisms from a set $\mathcal{X}$ to a set $\mathcal{Y}$ is the set of functions from $\mathcal{X}$ to $\mathcal{Y}$.
  In $\mathbf{Bnd}$, each morphism from a bound $\Gamma$ to another bound $\Gamma'$ is a permitted mapping $\theta$ from $\Gamma$ to $\Gamma'$.

We use a convenient shorthand for morphisms; $\mathcal{A} \xrightarrow{f} \mathcal{B}$ and $f : \mathcal{A} \to \mathcal{B}$ indicate that $f$ is a morphism from $\mathcal{A}$ to $\mathcal{B}$. A category also has two operations which make it more than just a multigraph. These operations are the key aspects of categories:

- $id$ assigns to each object $\mathcal{A}$ a special morphism in $\mathcal{M}or(\mathcal{A}, \mathcal{A})$, which we denote as $id_{\mathcal{A}}$.
  In $\mathbf{Set}$, the identity morphism for a set $\mathcal{X}$ is just the identify function on $\mathcal{X}$.
  In $\mathbf{Bnd}$, this requires that the set of permitted mappings from $\Gamma$ to $\Gamma$ includes the one mapping each variable to itself.
- Composition assigns to each chain of morphisms $\mathcal{A} \xrightarrow{f} \mathcal{B} \xrightarrow{g} \mathcal{C}$ a morphism directly from $\mathcal{A}$ to $\mathcal{C}$, which we denote as $\mathcal{A} \xrightarrow{f\,;g} \mathcal{C}$ or $\mathcal{A} \xrightarrow{g \circ f} \mathcal{C}$.
  In $\mathbf{Set}$, composition of morphisms is the standard composition of functions.
  In $\mathbf{Bnd}$, this essentially requires the set of permitted mappings to be closed under composition.

Lastly, these operations must satisfy two important properties:

- Identity morphisms must be identity elements for composition:
  $\forall f : \mathcal{A} \to \mathcal{B}.\ \ id_{\mathcal{A}}\,;f = f = f\,;id_{\mathcal{B}}$
- Composition must be associative:
  $\forall \mathcal{A} \xrightarrow{f} \mathcal{B} \xrightarrow{g} \mathcal{C} \xrightarrow{h} \mathcal{D}.\ \ (f\,;g)\,;h = f\,;(g\,;h)$

These properties obviously hold for $\mathbf{Set}$ as defined above. These properties hold for $\mathbf{Bnd}$ since it essentially uses composition of functions.

If we were to have just plain existential quantification of variables without constraints or expressions, then our category of bounds $\mathbf{Bnd}$ would look a lot like $\mathbf{Set}$ (although restricted to only finite sets). However, in our examples above we also had inheritance constraints such as $\alpha \ll \texttt{Shape}$ that needed to be preserved by the mapping $\theta$ for $\theta$ to be valid. For this category of bounds, the objects are more than just sets, they have some additional structure (in this case a binary relation), and the mappings must preserve this structure. Below we define $\mathbf{Rel}$, the category of binary relations and relation-preserving functions. The fact that morphisms are relation-preserving functions captures the fact that inheritance constraints such as $\alpha \ll \texttt{Shape}$ must be satisfied after substituting $\alpha$. $\mathbf{Rel}$ is defined as follows (proofs aside):

- An object $\mathcal{X}$ in $\mathbf{Rel}$ is a set $\mathcal{X}$ with a binary relation $\ll_{\mathcal{X}}$ on $\mathcal{X}$.
- A morphism $f$ in $\mathbf{Rel}$ from $\mathcal{X}$ to $\mathcal{Y}$ is a function $f : \mathcal{X} \to \mathcal{Y}$ mapping related elements to related elements:
  $\forall x, x' \in \mathcal{X}.\ \ x \ll_{\mathcal{X}} x' \Rightarrow f(x) \ll_{\mathcal{Y}} f(x')$.
- Identity and composition are the same as in $\mathbf{Set}$.

$\mathbf{Rel}$ has a *subcategory* that is particularly useful, say for modeling inheritance. $\mathbf{Prost}$ is the category of *preordered* sets and relation-preserving functions. $\mathbf{Prost}$ is like $\mathbf{Rel}$ but only contains objects $\mathcal{X}$ for which $\ll_{\mathcal{X}}$ is reflexive and transitive, hence it is a subcategory of $\mathbf{Rel}$. Later, this subcategory will be particularly important for formalizing subtyping and subtype-preserving substitutions.

In order to existentially quantify integers, we could use the category of finitely generated injective integer ring actions. Details aside, this category corresponds to allowing mappings of the form $i \mapsto c_1 * j + c_2$. The ring-action aspect allows the abstract domain to use the arithmetic equalities $c * (c_1 * j + c_2) = (c * c_1) * j + (c * c_2)$ and $(c_1 * j + c_2) + c = c_1 * j + (c_2 + c)$. This category is in fact the basis for our treatment of integers in our inferable typed assembly language [16], although with some additional structure for tracking unsigned 32-bit inequalities $<_{32+}$. Furthermore, we can take the product of two categories, such as the categories for classes with inheritance and for integer expressions with inequalities, in order to form a category for existential quantification over both classes and integers.

### 2.3 A Functor for Bodies

An existential type $\exists\Gamma.\tau$ has two parts: the bound $\Gamma$ and the body $\tau$. Above we just showed how can to formalize the bounds in an existential type system using a category. Here we formalize the bodies in an existential type system using a *functor*.

First we recognize that for a bound $\Gamma$ there may be many bodies $\tau$ such that $\exists\Gamma.\tau$ is a permitted existential type. Thus, for a context $\Gamma$, we have a set $Bodies(\Gamma)$ of permitted bodies for bound $\Gamma$. This is very much like the set of types which type check under some context. Furthermore, there is a subtyping relation $\leq_\Gamma$ on the set of bodies for $\Gamma$ because these bodies come from an abstract domain. This subtyping relation is reflexive and transitive (i.e. it is a preorder). Thus an existentially quantified abstract domain essentially specifies, for each bound $\Gamma$, a preordered set of

bodies permitted in bound $\Gamma$. In other words, $Bodies$ maps objects in $\mathbf{Bnd}$ to objects in $\mathbf{Prost}$; this forms the first component of a functor.

Second, we recognize that any permitted mapping $\theta$ from bound $\Gamma$ to bound $\Gamma'$ essentially specifies a substitution $[\theta]$ mapping bodies for $\Gamma$ to bodies for $\Gamma'$. This substitution replaces the $\Gamma$ variables in a body $\tau$ with the expressions they map to in terms of the $\Gamma'$ variables, so that $\tau[\theta]$ is a body using only variables in $\Gamma'$. Furthermore, if $\tau$ is a subtype of $\tau'$ in bound $\Gamma$, then $\tau[\theta]$ will be a subtype of $\tau'[\theta]$ in bound $\Gamma$. Thus $[\theta]$ is a relation-preserving function from the preordered set of bodies in $\Gamma$ to the preorder set of bodies in $\Gamma'$. In other words, the substitution operation $[-]$ maps each morphism $\theta : \Gamma \to \Gamma'$ in $\mathbf{Bnd}$ to a morphism $[\theta] : Bodies(\Gamma) \to Bodies(\Gamma')$ in $\mathbf{Prost}$; this forms the second component of a functor. Thus, we can formalize the informal concepts of bodies and substitution that we are already familiar with by using the categorical concept of functors.

Functors are like morphisms, but they go between categories rather than objects. A functor is comprised of two operations which must satisfy two properties. The operations of a functor $F$ from category $\mathbf{C}$ to category $\mathbf{D}$ are:

- A map $F$ from $\mathcal{O}bj_{\mathbf{C}}$ to $\mathcal{O}bj_{\mathbf{D}}$
- For each two objects $\mathcal{A}$ and $\mathcal{B}$ in $\mathbf{C}$, a map from $\mathcal{M}or_{\mathbf{C}}(\mathcal{A}, \mathcal{B})$ to $\mathcal{M}or_{\mathbf{D}}(F(\mathcal{A}), F(\mathcal{B}))$ (these maps are also denoted using $F$)

The operation $Bodies$ mapping each bound $\Gamma$ to a preordered set of permitted bodies for $\Gamma$, paired with the operation $[-]$ specifying for each permitted mapping $\theta : \Gamma \to \Gamma'$ a relation-preserving function $[\theta] : Bodies(\Gamma) \to Bodies(\Gamma')$, combine to form a functor $\langle Bodies, [-] \rangle$ from $\mathbf{Bnd}$ to $\mathbf{Prost}$. The operations of a functor must preserve the operational structure of categories by satisfying the following two properties:

- Preserves identities: $\forall \mathcal{A} \in \mathcal{O}bj_{\mathbf{C}}.\ \ F(id_{\mathcal{A}}) = id_{F(\mathcal{A})}$
  In particular, this means $[id_{\Gamma}]$ must be the identity function.
- Preserves composition: $\forall \mathcal{A} \xrightarrow{f} \mathcal{B} \xrightarrow{g} \mathcal{C}.\ F(f\,;g) = F(f)\,;F(g)$
  In particular, this means $\tau[\theta\,;\theta']$ must always equal $\tau[\theta][\theta']$.

### 2.4 Categorical Specification

Now that we have introduced the concepts of categories and functors and showed how they relate to existential quantification, we can formalize our representation of an existentially quantified abstract domain and existential subtyping. Our perspective is in line with that of op-indexed/fibred category theory [8], which demonstrates that our perspective properly formalizes the universal properties associated with existential quantification.

**Definition 1.** *An existentially quantified abstract domain is defined by two components:*

- *A category* $\mathbf{Bnd}$ *specifying bounds and permitted mappings*
- *A functor* $\langle Bodies, [-] \rangle$ *from* $\mathbf{Bnd}$ *to* $\mathbf{Prost}$ *specifying the preordered set of permitted bodies for each bound and the substitution for each permitted mapping*

*The elements of this existentially quantified abstract domain are of the form* $\exists \Gamma.\tau$ *where* $\Gamma$ *is an object of* $\mathbf{Bnd}$ *and* $\tau$ *is an element of* $Bodies(\Gamma)$*. The ordering on this existentially quantified abstract domain is defined by the following:*

$$\frac{\theta : \Gamma' \to \Gamma \text{ in } \mathbf{Bnd} \qquad \tau \leq_{\Gamma} \tau'[\theta] \ (\leq_{\Gamma} \text{ specified by } Bodies(\Gamma))}{\exists \Gamma.\tau \sqsubseteq \exists \Gamma'.\tau'}$$

Figure 1 gives an example specification of records with existentially quantified classes with inheritance. This existentially quantified abstract domain could be used as a crude system for modeling memory. The type $\exists \alpha.\langle\!|\alpha, \alpha, \alpha|\!\rangle$ represents a record with three fields, each of which is an instance of the same class $\alpha$. Prefix subtyping allows us to essentially forget that a field is present. In our inferable typed assembly language, we use records with prefix subtyping, but with much more expressive fields [16].

In our example, we do not use constants such as `Square`, generics, nor covariant arrays solely for the sake of easing the explanation of our framework. In the appendices, we show how to easily add all of these features and more. For example, we also show that rules such as $\forall \alpha.\alpha \ll \beta[] \Rightarrow \exists \gamma.\alpha = \gamma[] \wedge \gamma \ll \beta$, important for type checking the usage of arrays in C#, are compatible with our framework. The bounds we use in our inferable typed assembly language [16], including generic classes and arrays with multiple inheritance and simple integer expressions with

$\mathbf{Bnd}$: the subcategory of $\mathbf{Rel}$ containing only finite partial orders: $\langle \mathcal{V}_\Gamma, \ll_\Gamma \rangle$
$Bodies$: assigns $\Gamma$ to the set $List(\mathcal{V}_\Gamma)$ with $\ell \leq_\Gamma \ell'$ defined as $\ell'$ is a prefix of $\ell$
$[-]$: substitution for a mapping $\theta$ is simply $List.map(\theta)$

Example of existential subtypes: $\exists\alpha.\langle\!|\alpha, \alpha, \alpha|\!\rangle \quad \sqsubseteq \quad \exists\beta, \gamma : \beta \ll \gamma.\langle\!|\beta, \gamma|\!\rangle$
(constraints generated by reflexivity and transitivity are implicit)

**Fig. 1.** Specification for existentially quantified records of classes

unsigned 32-bit inequalities, was built and adapted using the simple design tools in the appendices. The appendices also includes significant generalizations of our framework as presented here. However, we use the simple example of records of classes with partially ordered inheritance in Figure 1 and our simplified framework because these ease the explanations and still capture the key concepts of our framework.

## 3 Deciding Existential Subtyping

Subtyping with existential quantification is generally undecidable [17]. For example, subtyping in the Scala programming language is undecidable [17]. Even subtyping in Java with wildcards, which are essentially a very restrictive form of existential quantification, is believed to be undecidable [2, 10, 17]. However, this does not mean subtyping with existential quantification must *always* be undecidable. Here we specify the requirements put forth by our framework for *general subtyping*. Using these requirements, we will show how the problem of existential subtyping can be reduced to whether one morphism *factors through* another morphism in the category $\mathbf{Bnd}$ of bounds and permitted mappings. In the section afterwards, we will show how these requirements can also be used to construct the join of two existential types. But, before we get ahead of ourselves, we will first discuss *skeletal* and *general* types, then we will show how to apply these simple concepts to existential subtyping.

### 3.1 Skeletal Types

Every body $\tau$ of an existential type $\exists\Gamma.\tau$ has a component which is independent of the bound $\Gamma$. That is, even as we apply substitutions to $\tau$ there is a component of the body will never change. This component is essentially the shape or *skeleton* of the body. For example, we can represent the skeleton of the body $\langle\!|\alpha, \beta, \beta|\!\rangle$ as the *skeletal type* $\langle\!|\star, \star, \star|\!\rangle$. That is to say that, no matter what substitutions we apply to $\langle\!|\alpha, \beta, \beta|\!\rangle$, it will always be a record with 3 fields. Informally we can construct the skeletal type of a body by replacing every expression dependent on the context with a star $\star$. Each $\star$ in a skeletal type essentially identifies an existentially quantifiable *slot*. Understanding and tracking these slots turns out to be essential for inferring existential types.

To formalize the simple but useful concept of skeletal types, we use the classic category theoretic concept of a *terminal object*. A terminal object in $\mathbf{Bnd}$ is a bound $\Gamma_\star$ with the property that every other bound $\Gamma$ has a *unique* morphism $!_\Gamma : \Gamma \to \Gamma_\star$ in $\mathbf{Bnd}$. The fact that this morphism always exists allows us to map *any* body $\tau$ for *any* bound $\Gamma$ to a body $\tau[!_\Gamma]$ for bound $\Gamma_\star$. Furthermore, the fact that this morphism is unique informs us that if we take a body $\tau$ for $\Gamma$, apply a substitution $\theta : \Gamma \to \Gamma'$ to get a body $\tau[\theta]$ for $\Gamma'$, and then map to $\Gamma_\star$ resulting in $\tau[\theta][!_{\Gamma'}]$, it would be just as if we had mapped $\tau$ to $\Gamma_\star$ *before* substituting with $\theta$. The proof is simple: $\tau[\theta][!_{\Gamma'}]$ equals $\tau[\theta\,;!_{\Gamma'}]$ since substitution is functorial, and $\theta\,;!_{\Gamma'}$ equals $!_\Gamma$ by uniqueness since both are morphisms from $\Gamma$ to $\Gamma_\star$, thus $\tau[\theta][!_{\Gamma'}] = \tau[\theta\,;!_{\Gamma'}] = \tau[!_\Gamma]$. We say the bodies for bound $\Gamma_\star$ are the skeletal types, and the operation $[!]$ (where the bound $\Gamma$ is implicit) is the starring process described above. The fact that $[!]$ produces the same result if applied before or after any other substitution formally captures the concept of substitution-invariance that skeletal types are intended to represent.

In our example of existentially quantified records, the terminal object $\Gamma_\star$ is $\langle \{\star\}, \{\star \ll_{\Gamma_\star} \star\} \rangle$. Because there is only one element, every other set has a unique function to $\{\star\}$ which is simply the constant function mapping everything to $\star$. Furthermore, because $\{\star \ll_{\Gamma_\star} \star\}$ is the maximum binary relation on $\{\star\}$, these functions will always be relation-preserving. The bodies for this context, that is the skeletal types, are simply lists of the singleton set $\{\star\}$. Since there is only one element, each list is classified entirely by its length. Thus subtyping $\ell \leq_{\Gamma_\star} \ell'$ is simply $length(\ell) \geq length(\ell')$. We call subtyping in $\Gamma_\star$ *skeletal subtyping*. Skeletal subtyping is much simpler than subtyping in general since all the quantifiable terms have been removed, and so it is much simpler for an analysis or type system

designer to reason about. For this reason, our framework reduces many of the more challenging properties down to skeletal subtyping and *representers*, which we discuss next. For brevity, we will represent the skeletal type of a body $\tau$ as $\tau_\star$ and skeletal subtyping as $\leq_\star$.

### 3.2 General Types

Just like every body has a skeletal type, every body also has a *general type*. This general type is best understood through skeletal types. In a skeletal type, each slot is represented by $\star$. To construct the general type, simply assign each slot its own variable. Thus, the general type for $\langle\star, \star, \star\rangle$ is $\exists\alpha, \beta, \gamma.\langle\alpha, \beta, \gamma\rangle$. We can do this for any body $\tau$: apply the substitution [!] to get the skeletal type $\tau_\star$ then replace each $\star$ in $\tau_\star$ with its own variable to get $\exists\Gamma_\tau.\tau_G$, the general type for $\tau$. We call $\Gamma_\tau$ the *general bound* for $\tau$ and $\tau_G$ the *general body* for $\tau$. Just like skeletal types, general types are substitution-invariant. In particular, any two bodies with the same skeletal type also have the same general type.

Once again, to formalize the simple but useful concept of general types, we return to category theory. Given a skeletal type $\tau_\star$, the general type $\exists\Gamma_{\tau_\star}.\tau_G$ has the property that, for *any* existential type $\exists\Gamma.\tau$ with skeletal type $\tau_\star$, there is a *unique* morphism $\mathit{rep}_\tau : \Gamma_{\tau_\star} \to \Gamma$ with the property that $\tau_G[\mathit{rep}_\tau]$ equals $\tau$. We call the morphism $\mathit{rep}_\tau$ the *representer* of $\tau$. The intuition is that the general type gives a name to each slot, and the representer specifies the value of each slot in $\tau$. Consider the following example:

| Original Type | General Type | Representer |
|---|---|---|
| $\exists\alpha, \beta : \alpha \ll \beta.\langle\alpha, \alpha, \beta\rangle$ | $\exists x, y, z.\langle x, y, z\rangle$ | $\{x, y \mapsto \alpha, z \mapsto \beta\}$ |

We use $x$, $y$, and $z$ in the general type to emphasize that it names each slot. The advantage of understanding skeletal types and general types is that we can now classify each existential type $\exists\Gamma.\tau$ by its bound $\Gamma$, its skeletal type $\tau_\star$, and its representer $\mathit{rep}_\tau : \Gamma_{\tau_\star} \to \Gamma$.

### 3.3 General Subtyping

So far we have simply been making observations of existential types, identifying that an existential type can be broken down into pieces, and formalizing each of these pieces categorically. Now we describe the first of the two fundamental requirements of our framework. We will see that this requirement also serves as a guideline, informing the abstract domain designer early on as to which features will be or may not be inferable.

We have already made a connection between skeletal *types* and general *types*. Now we want to make a connection between skeletal *subtyping* and general *subtyping*. That is, we require that whenever two skeletal types are subtypes then their general types are also existential subtypes. In other words, the following must hold:

$$\forall\tau_\star, \tau'_\star.\ \tau_\star \leq_\star \tau'_\star \implies \exists\Gamma_{\tau_\star}.\tau_G \sqsubseteq \exists\Gamma_{\tau'_\star}.\tau'_G$$

This means that for any skeletal subtypes $\tau_\star \leq_\star \tau'_\star$ there must be a morphism, which we call $\mathit{gen}_{\tau_\star \leq_\star \tau'_\star}$, from $\Gamma_{\tau'_\star}$ to $\Gamma_{\tau_\star}$ such that $\tau_G \leq_{\Gamma_{\tau_\star}} \tau'_G[\mathit{gen}_{\tau_\star \leq_\star \tau'_\star}]$ holds. We additionally require that $\mathit{gen}_{\tau_\star \leq_\star \tau'_\star}$ is the *unique* morphism with this property. The intuition is that, if two existential types are existential subtypes of each other, then it is because the slots in the body of the supertype somehow match up with the slots in the body of the subtype. For example, in our example of existentially quantified records with *prefix* subtyping, we have the following general subtyping:

$$\begin{array}{rcl} \langle\star, \star, \star\rangle & \leq_\star & \langle\star, \star\rangle \\ & \Downarrow & \\ \exists a, b, c.\langle a, b, c\rangle & \sqsubseteq & \exists x, y.\langle x, y\rangle \end{array}$$

with

$$\mathit{gen}_{\langle\star,\star,\star\rangle \leq_\star \langle\star,\star\rangle} = \{x \mapsto a, y \mapsto b\}$$

Had we used *suffix* subtyping instead, then $\mathit{gen}_{\langle\star,\star,\star\rangle \leq_\star \langle\star,\star\rangle}$ would be $\{x \mapsto b, y \mapsto c\}$ instead. Had we used *both* prefix and suffix subtyping, then there would be multiple candidates for $\mathit{gen}_{\langle\star,\star,\star\rangle \leq_\star \langle\star,\star\rangle}$, and we will see later why this is problematic for joining existential types.

There is one more formal requirement we must make to properly capture our concept of general subtyping. Our requirements above essentially say that the $\mathit{gen}$ morphisms classify subtyping in *general* bounds, but we want it to

classify subtyping in *all* bounds. That is, whenever $\tau \leq_\Gamma \tau'$ holds for bodies in any bound $\Gamma$, we want our *gen* morphisms to capture this subtyping as well. One observation is that whenever $\tau \leq_\Gamma \tau'$ holds then $\tau_\star \leq_\star \tau'_\star$ also holds since all substitution, particularly [!], preserve subtypes by assumption. Thus, we can construct the following *diagram*:

$$\begin{array}{ccc} \Gamma_\tau & \xleftarrow{gen_{\tau_\star \leq_\star \tau'_\star}} & \Gamma_{\tau'} \\ & {\scriptstyle rep_\tau} \searrow \quad \swarrow {\scriptstyle rep_{\tau'}} & \\ & \Gamma & \end{array}$$

Our requirement is that this diagram always *commutes*; composing morphisms along any path always produces the same result. In this case this means that $rep_\tau \circ gen_{\tau_\star \leq_\star \tau'_\star}$ equals $rep_{\tau'}$. Intuitively, this requirement says that, whenever $\tau$ is a subtype of $\tau'$ in bound $\Gamma$, then each slot that these two bodies have in common actually has the same value in both bodies. In other words, the representers for $\tau$ and $\tau'$ agree on these common slots identified by $gen_{\tau_\star \leq_\star \tau'_\star}$. A simple rule of thumb for general subtyping that makes for a good design guideline is that, for each syntactic subtype rule, all metavariables which occur in the supertype should also occur in the subtype.

This ends the formalization of our requirement of general subtyping. Next we see how we can apply this requirement to deciding existential subtyping. After that, we will give examples of why this requirement is important for joining existential types, or really examples of type systems that fail to meet this requirement and so also fail to have joins.

### 3.4 Abstract Algorithm for Existential Subtyping

Here we present our abstract algorithm for existential subtyping. We call this an abstract algorithm because it does not provide a decision procedure, but rather it provides an algorithm for reducing the problem of existential subtyping to a problem that is much simpler for many domains. First, we present our key theorem.

**Theorem 1.** *$\exists\Gamma.\tau \sqsubseteq \exists\Gamma'.\tau'$ holds if and only if $\tau_\star \leq_\star \tau'_\star$ holds and there exists a morphism $\theta : \Gamma' \to \Gamma$ such that the following diagram commutes:*

$$\begin{array}{ccc} \Gamma_\tau & \xleftarrow{gen_{\tau_\star \leq_\star \tau'_\star}} & \Gamma_{\tau'} \\ {\scriptstyle rep_\tau}\downarrow & & \downarrow{\scriptstyle rep_{\tau'}} \\ \Gamma & \xleftarrow[\theta]{} & \Gamma' \end{array}$$

*Proof.* By definition $\exists\Gamma.\tau \sqsubseteq \exists\Gamma'.\tau'$ holds if and only if there is a morphism $\theta : \Gamma' \to \Gamma$ such that $\tau \leq_\Gamma \tau'[\theta]$ holds. (For brevity, we will simply use *gen* for $gen_{\tau_\star \leq_\star \tau'_\star}$.)

$\Longrightarrow$: Assume a morphism $\theta : \Gamma' \to \Gamma$ exists such that $\tau \leq_\Gamma \tau'[\theta]$ holds. This implies that $\tau_\star \leq_\star \tau'_\star$ holds since $[!_\Gamma]$ preserves subtyping by requirement and $\tau'_\star$ equals $\tau'[\theta]_\star$ due to the substitution-invariance property of skeletal types proved earlier. By our last requirement for general subtyping, we know that $rep_\tau \circ gen$ equals $rep_{\tau'[\theta]}$ since $\tau \leq_\Gamma \tau'[\theta]$ holds. Thus we have the equalities $\tau'_G[rep_{\tau'[\theta]}] = \tau'[\theta]_G[rep_{\tau'[\theta]}] = \tau'[\theta] = \tau'_G[rep_{\tau'}][\theta] = \tau'_G[rep_{\tau'}; \theta]$, therefore $rep_{\tau'[\theta]}$ equals $rep_{\tau'}; \theta$ by the uniqueness requirement for representers. And so the diagram commutes because we have $gen_{\tau_\star \leq_\star \tau'_\star}; rep_\tau = rep_\tau \circ gen = rep_{\tau'[\theta]} = rep_{\tau'}; \theta$.

$\Longleftarrow$: Assume $\tau_\star \leq_\star \tau'_\star$ holds and there exists a morphism $\theta : \Gamma' \to \Gamma$ such that the diagram commutes. By requirement for general subtyping, we know that $\tau_G \leq_{\Gamma_\tau} \tau'_G[gen]$ holds. By requirement $[rep_\tau]$ preserves subtypes, so we also know that $\tau_G[rep_\tau] \leq_\Gamma \tau'_G[gen][rep_\tau]$ holds. By requirement, $\tau_G[rep_\tau]$ equals $\tau$. By assumption $gen; rep_\tau$ equals $rep_{\tau'}; \theta$, so we have the equalities $\tau'_G[gen][rep_\tau] = \tau'_G[gen; rep_\tau] = \tau'_G[rep_{\tau'}; \theta] = \tau'_G[rep_{\tau'}][\theta]$, and the last equals $\tau'[\theta]$ by requirement for representers. By substitution, we therefore have that $\tau \leq_\Gamma \tau'[\theta]$ holds.

If such a $\theta$ exists, then $rep_\tau \circ gen_{\tau_\star \leq_\star \tau'_\star}$ is said to *factor through* $rep_{\tau'}$. For many categories, determining whether one morphism factors through another is decidable. In our example of records with existentially quantified classes with inheritance, determining whether a constraint $\alpha \ll \beta$ is preserved breaks down to graph reachability. In an analysis for advanced array-bounds elimination, this question may require a decision procedure for convex polygon containment in modular arithmetic. Regardless, the above theorem allows us to reduce the challenge of existential subtyping to using these standard domain-specific decision procedures.

Our abstract algorithm for deciding existential subtyping is defined as follows using the result of our theorem:

DECIDEEXISTENTIALSUBTYPES($\exists\Gamma.\tau$, $\exists\Gamma'.\tau'$):
  if ($\tau[!] \leq_\star \tau'[!]$) then
    construct $\mathit{gen}_{\tau[!]\leq_\star\tau'[!]}$, $\mathit{rep}_\tau$, and $\mathit{rep}_{\tau'}$;
    return $\mathit{rep}_\tau \circ \mathit{gen}_{\tau[!]\leq_\star\tau'[!]}$ *factors through* $\mathit{rep}_{\tau'}$;
  else return **false**;

In our experience [16], this algorithm is best implemented in two phases. Construct $\mathit{rep}_\tau \circ \mathit{gen}_{\tau[!]\leq_\star\tau'[!]}$ and $\mathit{rep}_{\tau'}$ *while deciding* $\tau[!] \leq_\star \tau'[!]$. If that process succeeds, use the decision procedure to determine whether $\mathit{rep}_\tau \circ \mathit{gen}_{\tau[!]\leq_\star\tau'[!]}$ factors through $\mathit{rep}_{\tau'}$.

To understand the algorithm, consider the following examples:

$$(1) \qquad \exists\alpha.\langle\alpha\rangle \stackrel{?}{\sqsubseteq} \exists\delta,\varepsilon.\langle\delta,\varepsilon\rangle$$
$$(2) \quad \exists\alpha,\beta:\alpha\ll\beta.\langle\alpha,\beta\rangle \stackrel{?}{\sqsubseteq} \exists\delta.\langle\delta,\delta\rangle$$
$$(3) \qquad \exists\alpha,\beta.\langle\alpha,\beta\rangle \stackrel{?}{\sqsubseteq} \exists\delta,\varepsilon:\delta\ll\varepsilon.\langle\delta,\varepsilon\rangle$$
$$(4) \qquad \exists\alpha.\langle\alpha,\alpha,\alpha\rangle \stackrel{?}{\sqsubseteq} \exists\delta,\varepsilon:\delta\ll\varepsilon.\langle\delta,\varepsilon\rangle$$

Example (1) fails because $\langle\star\rangle$ is not a skeletal subtype of $\langle\star,\star\rangle$. Example (2) fails because the function $\{x\mapsto\alpha, y\mapsto\beta\}$ does not factor through the function $\{x,y\mapsto\delta\}$. Another way to think about it is that $\{x\mapsto\alpha, y\mapsto\beta\}$ fails to satisfy the implicit equational constraints of $\{x,y\mapsto\delta\}$. Example (3) fails because, even though the function $\{x\mapsto\alpha, y\mapsto\beta\}$ does factor through the function $\{x\mapsto\delta, y\mapsto\varepsilon\}$ via the function $\{\delta\mapsto\alpha,\gamma\mapsto\beta\}$, this function $\{\delta\mapsto\alpha,\gamma\mapsto\beta\}$ fails to preserve the relational constraint $\delta\ll\varepsilon$ and so is not a *morphism* in our category of *relation-preserving* functions. Example (4) succeeds via the function $\{\delta,\varepsilon\mapsto\alpha\}$ which preserves the relational constraint $\delta\ll\varepsilon$ because partial orders are always reflexive (so $\alpha\ll\alpha$ is implicitly present in the subtype).

### 3.5 A Need for General Subtyping

We have no proof, or even belief, that general subtyping is truly necessary for an existential type system to have joins, given the bizarre abstract domains that can be constructed. However, it has been our experience that any extension we would add that did not satisfy our requirement of general subtyping would actually produce an existential type system that fails to have joins. Here we present one such example: null-pointer types. We choose this example because we spent a *lot* of time trying to extend our framework to handle null-pointer types because it seemed so obvious that they should work. However, after a lot of suffering we finally realized that there actually *are no joins* with standard null-pointer types. Thus be relaying the lessons we learned we hope to save other existentially-quantified-abstract-domain designers from the same suffering we experienced.

Null-pointer types may be one of the most obvious extensions to add to our existentially quantified records with classes and inheritance; replacing any field of a record with the null-pointer type produces a subtype of that record, assuming we are using covariant record types. In particular, we have the following subtype rule:

$$\frac{\omega \text{ can be any term specifying a class in bound } \Gamma}{\langle\texttt{null}\rangle \leq_\Gamma \langle\omega\rangle}$$

However, this rule does not fit within our framework. It specifies that the skeletal subtyping $\langle\texttt{null}\rangle \leq_\star \langle\star\rangle$ should hold. However, the existential subtyping $\exists\varnothing.\langle\texttt{null}\rangle \sqsubseteq \exists x.\langle x\rangle$ does not hold: there is nothing to map $x$ to. One might try to fix this by adding constants, then map $x$ could map to any one of these constants, but then $\mathit{gen}_{\langle\texttt{null}\rangle\leq_\star\langle\star\rangle}$ is no longer unique. One might try to fix this again by specifying that there is only one constant, say `Object`, so that $\mathit{gen}_{\langle\texttt{null}\rangle\leq_\star\langle\star\rangle}$ would be unique. However this would still not satisfy the last requirement of general subtyping: for any $\tau \leq_\Gamma \tau'$, $\mathit{rep}_\tau \circ \mathit{gen}_{\tau_\star\leq_\star\tau'_\star}$ must always equal $\mathit{rep}_{\tau'}$. In particular, we would have the following counterexample:

$$\frac{\text{Subtypes}}{\langle\texttt{null}\rangle \leq_{\{\alpha\}} \langle\alpha\rangle} \quad \frac{\mathit{rep}_\tau \circ \mathit{gen}_{\tau_\star\leq_\star\tau'_\star}}{\{x\mapsto\texttt{Object}\}} \quad \frac{\mathit{rep}_{\tau'}}{\{x\mapsto\alpha\}}$$

This may seem like a limitation of our framework, as we first suspected, but in fact there are no joins with the standard null-pointer type. Fortunately, we can concisely demonstrate this fact here. Consider the following two existentially quantified records:

$$R_\alpha = \exists\alpha.\langle\texttt{null},\alpha,\texttt{null}\rangle \text{ and } R_\beta = \exists\beta,\beta'.\langle\beta,\texttt{null},\beta'\rangle$$

They have the following two common existential supertypes:

$$R_\gamma = \exists\gamma, \gamma'.\langle\gamma, \gamma, \gamma'\rangle \text{ and } R_\delta = \exists\delta, \delta'.\langle\delta, \delta', \delta'\rangle$$

$R_\gamma$ is not an existential subtype of $R_\delta$, nor vice-versa. However, it is easy to see that, should $R_\alpha$ and $R_\beta$ have a join, that join must have skeletal type $\langle\star, \star, \star\rangle$ with no `null` field. The only common existential subtype of $R_\gamma$ and $R_\delta$ with that skeleton is $R_\varepsilon = \exists\varepsilon.\langle\varepsilon, \varepsilon, \varepsilon\rangle$. However, $R_\varepsilon$ is not an existential supertype of $R_\beta$, so $R_\alpha$ and $R_\beta$ have no join. Thus, abstract interpretation could only produce *incomplete* analyses using this abstract domain.

This example demonstrates how useful our framework can be to someone designing an existentially quantified abstract domain. By following our simple guidelines for general subtyping, the designer is saved from attempting to add seemingly simple but ultimately damaging types such as null-pointer types. In the above example, the fundamental problem is that it is ambiguous whether the `null` in $R_\beta$ should correspond to $\beta$, $\beta'$, or even some other class constant. In our typed assembly language, we used our framework's guidelines to determine that inference would be possible if each null-pointer type were annotated with the class it is meant to be a null pointer to, since this adaptation would satisfy the general subtyping requirement. This is the only intraprocedural annotation requirement of our typed assembly language, but as we just demonstrated it is an unfortunate but unavoidable cost.

## 4 Joining Existential Types

Joining existential types is quite challenging. We just showed that even with simple null-pointer types joining existential types is actually impossible. In the last section we defined the first fundamental requirement of our framework: general subtyping. General subtyping essentially allows us to track the values of existentially quantifiable slots through the subtype system. This alone does not solve our problem though. Consider the following two records (temporarily adding constants to our example type system):

$$\exists\emptyset.\langle\texttt{Square}, \texttt{Square}, \texttt{Rhombus}\rangle \text{ and } \exists\emptyset.\langle\texttt{Circle}, \texttt{Oval}, \texttt{Oval}\rangle$$

One might expect that because these use only constants their join would be simple, but in fact their join is surprisingly complex:

$$\exists\alpha, \beta, \gamma : \alpha \ll \beta \ll \gamma.\langle\alpha, \beta, \gamma\rangle$$

Not only does a join algorithm need to recognize that, even though there are only two constants in each type it still needs three variables to represent them because the constants do not line up, but also that there is a left-to-right inheritance structure because of reflexivity. In this section we present the second fundamental requirement of our framework and then apply that requirement to define our abstract algorithm for joining types. But before that, let us examine what tools and information we already have available to us.

Suppose we have two existential types we want to join: $\exists\Gamma_1.\tau^1$ and $\exists\Gamma_2.\tau^2$. By using skeletal types, we can first operate in a much simpler space and construct their skeletal join $\tau_\star^\sqcup$ as $\tau_\star^1 \sqcup_\star \tau_\star^2$. We can then use general subtyping and representers to construct the diagram in Figure 2. Recall that the general bound $\Gamma_{\tau_\star^\sqcup}$ essentially assigns a separate name to each slot. Thus the morphisms in Figure 2 are maps from the named slots in $\tau_\star^\sqcup$ to their originating values defined in $\Gamma_1$ and $\Gamma_2$. In our earlier example, the generalized type for the skeletal join would be $\exists x, y, z.\langle x, y, z\rangle$ and we would have maps $\{x, y \mapsto \texttt{Square}, z \mapsto \texttt{Rhombus}\}$ and $\{x \mapsto \texttt{Circle}, y, z \mapsto \texttt{Oval}\}$. What we need to construct, then, is a bound which constrains $x$, $y$, and $z$ *as much as possible*, possibly even merging them, while still having these maps be valid. Graphically, we need to construct a *maximally constrained* bound $\Gamma_\sqcup$ with morphisms $\theta_1^\sqcup$ and $\theta_2^\sqcup$ so that the diagram in Figure 3. The morphism $r$ describes how to map the general bound $\Gamma_{\tau_\star^\sqcup}$ to the maximally constrained bound $\Gamma_\sqcup$. We can use $r$ to convert our general body for the skeletal join $\tau_\star^\sqcup$ into the body $\tau^\sqcup = \tau_G^\sqcup[r]$. The resulting type $\exists\Gamma_\sqcup.\tau^\sqcup$ is the join of $\exists\Gamma_1.\tau^1$ and $\exists\Gamma_2.\tau^2$.

In our example, the maximally constrained bound $\Gamma_\sqcup$ would be $\alpha, \beta, \gamma : \alpha \ll \beta \ll \gamma$. The morphism $r$ would be the mapping $\{x \mapsto \alpha, y \mapsto \beta, z \mapsto \gamma\}$. The body $\tau^\sqcup$ for bound $\Gamma_\sqcup$ would be $\langle x, y, z\rangle[x \mapsto \alpha, y \mapsto \beta, z \mapsto \gamma]$ which evaluates to $\langle\alpha, \beta, \gamma\rangle$. Put it all together and we determine that our join type $\exists\Gamma_\sqcup.\tau^\sqcup$ is $\exists\alpha, \beta, \gamma : \alpha \ll \beta \ll \gamma.\langle\alpha, \beta, \gamma\rangle$ as we presented earlier.

The only piece missing in order to join existential types algorithmically is a technique for constructing the maximally constrained bound $\Gamma_\sqcup$. For this, we use a categorical concept called *factorization structures* [1]. A factorization structure will simultaneously provide the maximally constrained bound $\Gamma_\sqcup$ and the morphisms $r$, $\theta_1^\sqcup$, and $\theta_2^\sqcup$. The proof that $\exists\Gamma_\sqcup.\tau^\sqcup$ is a join will be concise and direct. The need for a factorization structure is the second of the two

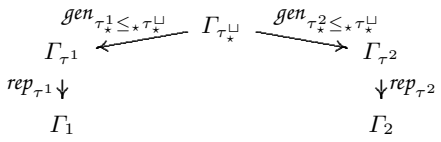



**Fig. 2.** Morphisms available due to general subtyping

fundamental requirements of our framework. Note that this section is entirely in terms of the bounds of existential types, and makes no mention of the bodies. The rule of general subtyping in the prior section is the only constraint on bodies, so one is free to design bodies without any concern for the concepts in this section. Before defining our join algorithm, we must define what factorization structures are, beginning with the categorical concept of *sources*.

### 4.1 Sources

A diagram of the form $\mathcal{B}_1 \xleftarrow{f_1} \mathcal{A} \xrightarrow{f_2} \mathcal{B}_2$ is called a 2-source: it is comprised of two morphisms with the same domain. In particular, the diagram in Figure 2 is a 2-source. There are many special kinds of sources, such as *mono-sources* [1]. Consider the 2-source $\mathcal{X}_1 \xleftarrow{\pi_1} \mathcal{X}_1 \times \mathcal{X}_2 \xrightarrow{\pi_2} \mathcal{X}_2$ in $\mathbf{Set}$. It has the property that for any two elements $p$ and $p'$ of $\mathcal{X}_1 \times \mathcal{X}_2$, if $\pi_1(p)$ equals $\pi_1(p')$ and $\pi_2(p)$ equals $\pi_2(p')$ then $p$ must equal $p'$. This makes $(\mathcal{X}_1 \times \mathcal{X}_2 \xrightarrow{\pi_i} \mathcal{X}_i)_{i \in \{1,2\}}$ a mono-source in $\mathbf{Set}$. More generally, a mono-source $(\mathcal{A} \xrightarrow{m_i} \mathcal{B}_i)_{i \in \mathcal{I}}$ has the property that if two morphisms $f, g : \mathcal{C} \to \mathcal{A}$ are equal after composing with $m_i$ for each $i$ in $\mathcal{I}$, then $f$ must equal $g$: $\forall f, g : \mathcal{C} \to \mathcal{A}.\ (\forall i \in \mathcal{I}.\, f\,;m_i = g\,;m_i) \Rightarrow f = g$. The mono-1-sources in $\mathbf{Set}$ are precisely the injective functions.

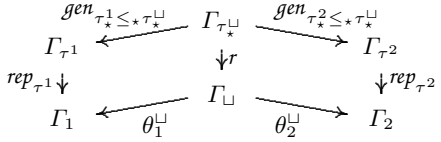



**Fig. 3.** Diagram for the maximally constrained bound $\Gamma_\sqcup$

### 4.2 Factorizations

The category $\mathbf{Set}$ has the interesting property that any source $(\mathcal{X} \xrightarrow{f_i} \mathcal{Z}_i)_{i \in \mathcal{I}}$ can be *factored* into $\mathcal{X} \xrightarrow{e} (\mathcal{Y} \xrightarrow{m_i} \mathcal{Z}_i)_{i \in \mathcal{I}}$ where $e$ is a surjection and $(\mathcal{Y} \xrightarrow{m_i} \mathcal{Z}_i)_{i \in \mathcal{I}}$ is a mono-source. Factoring here means that $e\,;m_i$ equals $f_i$ for all $i$ in $\mathcal{I}$. To do this, we construct an equivalence relation $\approx$ on $\mathcal{X}$, defined by $x$ and $x'$ are equivalent when $f_i(x)$ equals $f_i(x')$ for all $i$ in $\mathcal{I}$. We then define $\mathcal{Y}$ as $\mathcal{X}/\approx$, the set of all equivalence classes. $e$ is the function mapping each element $x$ to its equivalence class $[x]_\approx$. $m_i$ is the function mapping $[x]_\approx$ to $f_i(x)$, which is well-defined by construction of $\approx$. In a way, we are merging the elements of $\mathcal{X}$ as much as possible for each $m_i$ to still be a well-defined function. Thus, $\mathcal{Y}$ is $\mathcal{X}$ maximally constrained by equalities such that each $m_i$ is still a well-defined function, tieing back to our strategy for joining existential types.

Since any source in $\mathbf{Set}$ can factor into a surjection and a mono-source, $\mathbf{Set}$ is said to have (Surjection, Mono-Source)-factorizations. More generally, a category $\mathbf{C}$ has $(\mathcal{E}, \mathcal{M})$-factorizations, where $\mathcal{E}$ is a class of morphisms and $\mathcal{M}$ is a class of sources, if any source in $\mathbf{C}$ can be factored into $\mathcal{A} \xrightarrow{e} (\mathcal{B} \xrightarrow{m_i} \mathcal{C}_i)_{i \in \mathcal{I}}$ where $e$ belongs to $\mathcal{E}$ and $(\mathcal{B} \xrightarrow{m_i} \mathcal{C}_i)_{i \in \mathcal{I}}$ belongs to $\mathcal{M}$ [1].

### 4.3 Diagonalizations

The category $\mathbf{Set}$ has yet another property which we will need to prove our join construction correct. Suppose we have sets and functions such that the following diagram commutes for all $i$ in $\mathcal{I}$:

$$\begin{array}{lll} \mathcal{W} \xrightarrow{e} \mathcal{X} & & \text{(where } e \text{ is a surjection)} \\ f\downarrow \qquad \downarrow g_i & & \\ \mathcal{Y} \underset{m_i}{\rightrightarrows} \mathcal{Z}_i & & \text{(and } (\mathcal{Y} \xrightarrow{m_i} \mathcal{Z}_i)_{i \in \mathcal{I}} \text{ is a mono-source)} \end{array}$$

In this situation, there will always be a unique function $d : \mathcal{X} \to \mathcal{Y}$ which cuts the square diagonally and commutes for all $i$ in $\mathcal{I}$. In fact, $d$ is easy to construct. Since $e$ is surjective, every value in $\mathcal{X}$ is of the form $e(w)$ where $w$ is an element of $\mathcal{W}$. We define $d$ as $d(e(w)) = f(w)$. To prove that $d$ is well-defined, we have to show that $e(w) = e(w')$ implies $f(w) = f(w')$. Assuming $e(w) = e(w')$, then $g_i(e(w)) = g_i(e(w'))$ holds for all $i$ in $\mathcal{I}$. Since we assumed the diagram always commutes, this implies $m_i(f(w)) = g_i(e(w)) = g_i(e(w')) = m_i(f(w'))$ for all $i$ in $\mathcal{I}$. But we also assumed $(\mathcal{Y} \xrightarrow{m_i} \mathcal{Z}_i)_{i \in \mathcal{I}}$ is a mono-source, so these equalities imply that $f(w)$ actually equals $f(w')$, which is our desired conclusion.

Because such a $d$ always exists uniquely in any such situation, we say that $\mathbf{Set}$ has unique (Surjection, Mono-Source)-diagonalizations. More generally, a category $\mathbf{C}$ is said to have unique $(\mathcal{E}, \mathcal{M})$-diagonalizations [1] if $e : \mathcal{A} \to \mathcal{B}$ belongs to $\mathcal{E}$ and $(\mathcal{C} \xrightarrow{m_i} \mathcal{D}_i)_{i \in \mathcal{I}}$ belongs to $\mathcal{M}$ implies the following for any morphism $f : \mathcal{A} \to \mathcal{C}$ and source $(\mathcal{B} \xrightarrow{g_i} \mathcal{D}_i)_{i \in \mathcal{I}}$:

$$\left(\begin{array}{ccc} & \mathcal{A} \xrightarrow{e} \mathcal{B} & \\ \forall i \in \mathcal{I}. & f\downarrow \qquad \downarrow g_i & \\ & \mathcal{C} \underset{m_i}{\rightrightarrows} \mathcal{D}_i & \\ & \textit{commutes} & \end{array}\right) \begin{array}{c} \textit{exists} \\ \textit{unique} \\ d : \mathcal{B} \to \mathcal{C} \\ \textit{such that} \\ \Longrightarrow \end{array} \left(\begin{array}{ccc} & \mathcal{A} \xrightarrow{e} \mathcal{B} & \\ \forall i \in \mathcal{I}. & f\downarrow \overset{d}{\swarrow} \downarrow g_i & \\ & \mathcal{C} \underset{m_i}{\rightrightarrows} \mathcal{D}_i & \\ & \textit{commutes} & \end{array}\right)$$

### 4.4 Factorization Structures

A category $\mathbf{C}$ is said to have an $(\mathcal{E}, \mathcal{M})$ factorization structure if it has both $(\mathcal{E}, \mathcal{M})$-factorizations and unique $(\mathcal{E}, \mathcal{M})$-diagonalizations [1]. The factorizations capture the ability to add constraints, and the unique diagonalizations capture the fact that these constraints are added maximally. Therefore, we require the category $\mathbf{Bnd}$ to have some chosen factorization structure which will be used to construct the maximally constrained bound when joining existential types. Although factorization structures are an unfamiliar concept, they are very common and the appendices shows a variety of techniques for building custom categories with powerful factorization structures.

Here we will define the factorization structure we will use for our example of existentially quantified classes with inheritance. Remember this example category $\mathbf{Bnd}$ is the subcategory of $\mathbf{Rel}$ satisfying reflexivity, transitivity, and anti-symmetry. It turns out that reflexivity, transitivity, and anti-symmetry can all be expressed as surjective categorical *implications* [1]. Because of this we will show that $\mathbf{Rel}$ has a (Surjection, Initial Mono-Source) factorization structure, and [1] informs us that our example category $\mathbf{Bnd}$ automatically inherits this same factorization structure (this is one of the techniques explained in more detail in the appendices). But first, we will define what an initial mono-source in $\mathbf{Rel}$ is.

An initial mono-source [1] $(\mathcal{A} \xrightarrow{m_i} \mathcal{B}_i)_{i \in \mathcal{I}}$ in $\mathbf{Rel}$ is a source whose underlying source $(\mathcal{A} \xrightarrow{m_i} \mathcal{B}_i)_{i \in \mathcal{I}}$ is a mono-source in $\mathbf{Set}$ with the additional property that, for any two elements $a$ and $a'$ in $\mathcal{A}$, $a \ll_{\mathcal{A}} a'$ holds if and only if $m_i(a) \ll_{\mathcal{B}_i} m_i(a')$ holds for all $i$ in $\mathcal{I}$. Essentially, $\ll_{\mathcal{A}}$ is the strongest binary relation on $\mathcal{A}$ such that each $m_i$ is still relation-preserving. Once again, this connects to our goal of constructing a maximally constrained bound $\Gamma_{\sqcup}$.

The first component of a factorization structure is factorizations, so must construct (Surjection, Initial Mono-Source)-factorizations in $\mathbf{Rel}$. Given a source $(\mathcal{A} \xrightarrow{f_i} \mathcal{C}_i)_{i \in \mathcal{I}}$, we start by constructing the (Surjection, Mono-Source)-factorization of the underlying source in $\mathbf{Set}$: $\mathcal{A} \xrightarrow{e} (\mathcal{B} \xrightarrow{m_i} \mathcal{C}_i)_{i \in \mathcal{I}}$. Remember this process constructs $\mathcal{B}$ by essentially merging as many elements of $\mathcal{A}$ as possible for each $m_i$ to still be a well-defined function (i.e. imposes as many equational constraints as possible). Next we define $b \ll_{\mathcal{B}} b'$ as $\forall i \in \mathcal{I}.\ m_i(b) \ll_{\mathcal{C}_i} m_i(b')$, the strongest binary relation on $\mathcal{B}$ such that each $m_i$ is relation-preserving. Thus by construction, each function $m_i$ is a morphism $m_i : \mathcal{B} \to \mathcal{C}_i$ in $\mathbf{Rel}$ and together form an initial mono-source. Lastly, we need to prove that $e$ is a morphism $e : \mathcal{A} \to \mathcal{B}$ in $\mathbf{Rel}$.

This is true precisely when $a \ll_{\mathcal{A}} a'$ implies $e(a) \ll_{\mathcal{B}} e(a')$ for all $a$ and $a'$ in $\mathcal{A}$. Assuming $a \ll_{\mathcal{A}} a'$ holds, we know $f_i(a) \ll_{C_i} f_i(a')$ holds for all $i$ in $\mathcal{I}$ since each $f_i$ is a morphism in $\mathbf{Rel}$. By construction, this implies that $m_i(e(a)) = f_i(a) \ll_{C_i} f_i(a') = m_i(e(a'))$ holds for all $i$ in $\mathcal{I}$. But this is precisely the definition of $e(a) \ll_{\mathcal{B}} e(a')$, so $e$ is a morphism in $\mathbf{Rel}$. By construction, $e$ is surjective and $(\mathcal{B} \xrightarrow{m_i} C_i)_{i \in \ll I}$ is an initial mono-source. Thus, $\mathbf{Rel}$ has (Surjection, Initial Mono-Source)-factorizations.

Second, we have to construct unique (Surjection, Initial Mono-Source)-diagonalizations. Suppose we have the appropriate commuting squares. The underlying commuting squares in $\mathbf{Set}$ have a surjection and a mono-source. So, we can construct $d : \mathcal{B} \to \mathcal{C}$ as the uniquely induced diagonal for the underlying commuting squares. We still need to prove that $d$ is a morphism $d : \mathcal{B} \to \mathcal{C}$ in $\mathbf{Rel}$. Assume we have elements $b$ and $b'$ in $\mathcal{B}$ such that $b \ll_{\mathcal{B}} b'$ holds. Since each $g_i$ is a morphism in $\mathbf{Rel}$, we know $g_i(b) \ll_{\mathcal{D}_i} g_i(b')$ holds for all $i$ in $\mathcal{I}$. By construction of $d$, this informs us that $m_i(d(b)) = g_i(b) \ll_{\mathcal{D}_i} g_i(b') = m_i(d(b'))$ holds for all $i$ in $\mathcal{I}$. Since $(\mathcal{C} \xrightarrow{m_i} \mathcal{D}_i)_{i \in \mathcal{I}}$ is an initial mono-source, by definition of initial mono-source this implies that $d(b) \ll_{\mathcal{C}} d(b')$ holds. Thus, $d : \mathcal{B} \to \mathcal{C}$ is a unique diagonal in $\mathbf{Rel}$. The remaining requirements are easy to prove from the definition of $d$.

Thus, $\mathbf{Set}$ has a (Surjection, Mono-Source) factorization structure, $\mathbf{Rel}$ has a (Surjection, Initial Mono-Source) factorization structure, and our example category $\mathbf{Bnd}$ for classes with inheritance has the same factorization structure. Factorization structures are common, and even a single category can have many factorization structures [1]. The ones above are fairly general purpose factorization structures, but factorization structures can also be constructed for specialized domains. In order to check array bounds in our typed assembly language, we built a factorization structure for integer expressions of the form $c_1 * i + c_2$ [16]. Different factorization structures have different benefits. Some can be easier to compute while others can be more expressive.

Next we discuss *tight* existential types for a given factorization structure. This concept gives some direction as to which factorization structure to choose for a given category of bounds because our join algorithm will require that all existential types be tight.

### 4.5 Tight Existential Types

In order to use our framework, we require that all existential types be *tight*. This is a necessary requirement, since it is easy to construct existential types without joins if non-tight existential types are permitted. Intuitively, a tight existential type is one for which all variables in the bound are used in the body; thus, there are no loose variables in the bound that should be dropped. Formally, a tight existential type is defined in terms of the $(\mathcal{E}, \mathcal{M})$ factorization structure chosen for $\mathbf{Bnd}$. Remember that each body $\tau$ in bound $\Gamma$ has a general type $\exists \Gamma_\tau.\tau_G$ and a representer $rep_\tau : \Gamma_\tau \to \Gamma$.

**Definition 2.** *An existential type $\exists \Gamma.\tau$ is tight for an $(\mathcal{E}, \mathcal{M})$ factorization structure if its representer $rep_\tau$ belongs to $\mathcal{E}$.*

If we use the (Surjection, –) factorization structure of $\mathbf{Set}$, $\mathbf{Rel}$, or our example $\mathbf{Bnd}$ for classes with inheritance, this means that each variable $\alpha$ in bound $\Gamma$ must be mapped to by some variable in $\Gamma_\tau$, which happens precisely when $\alpha$ occurs as the value of some slot in $\tau$, matching the intuition described above.

This requirement has an important impact on the choice of factorization structure for $\mathbf{Bnd}$. In particular, the larger $\mathcal{E}$ is, the more tight existential types there are, so the more precise the existentially quantified abstract domain can be. The largest $\mathcal{E}$ can be is the class of *epimorphisms* [1]. The epimorphisms in $\mathbf{Set}$, $\mathbf{Rel}$, and our example $\mathbf{Bnd}$ are precisely the surjections, so the factorization structures we defined earlier are optimal for precision. In general, an epimorphism $e : \mathcal{A} \to \mathcal{B}$ is a morphism with the property that if $e\,;f$ equals $e\,;g$ then $f$ must equal $g$. Also, since for any factorization structure all elements of $\mathcal{E}$ are epimorphisms [1], a morphism $\theta$ evidencing an existential subtyping between tight existential types is necessarily unique.

During abstract interpretation, a non-tight existential type may be constructed by the monotonic dataflow function. Fortunately, there is an optimal way to *tighten* an existential type $\exists \Gamma.\tau$. Simply factor $rep_\tau$ (viewed as a 1-source) into an $\mathcal{E}$-$\mathcal{M}$ sequence $\Gamma_\tau \xrightarrow{e} \Gamma' \xrightarrow{m} \Gamma$. $m$ serves as evidence that $\exists \Gamma.\tau$ is an existential subtype of $\exists \Gamma'.\tau_G[e]$, and the diagonalization property can be used to show that any tight existential supertype of $\exists \Gamma.\tau$ is also an existential supertype of $\exists \Gamma'.\tau_G[e]$. In our example with class inheritance, tightening would proceed as in the following:

$$\exists \alpha, \beta, \gamma : \alpha \ll \beta \ll \gamma.\langle \alpha, \beta, \beta \rangle \xmapsto{\textit{tightens to}} \exists \alpha, \beta : \alpha \ll \beta.\langle \alpha, \beta, \beta \rangle$$

The bound variable $\gamma$ is dropped, along with all constraints pertaining to $\gamma$, because it does not occur in the body. Although tightening may lose information, as much information is retained as possible, and often the lost information is unwanted anyways (such as bound variables which do not actually occur in the body).

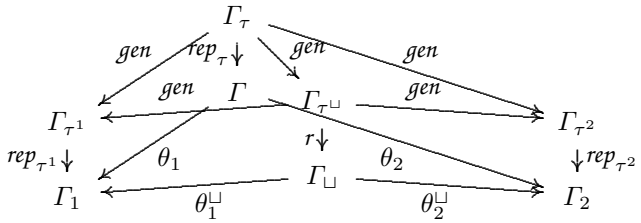
### 4.6 Abstract Algorithm for Joining Existential Types

At last we have all of the components necessary for joining existential types. We begin with our key theorem which defines the construction of the join and proves it correct.

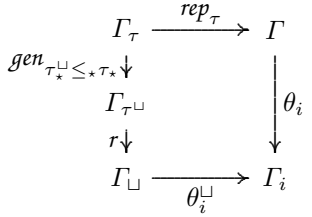
**Theorem 2.** *If* $\mathbf{Bnd}$ *has an* $(\mathcal{E}, \mathcal{M})$ *factorization structure, then given two existential types* $\exists \Gamma_1.\tau^1$ *and* $\exists \Gamma_2.\tau^2$*, suppose* $\tau^{\sqcup}_{\star}$ *is the skeletal join of the skeletal types* $\tau^1_\star$ *and* $\tau^2_\star$*, and the 2-source* $(\Gamma_{\tau^{\sqcup}_{\star}} \xrightarrow{gen_{\tau^i_\star \leq_\star \tau^{\sqcup}_\star}} \Gamma_{\tau^i} \xrightarrow{rep_{\tau^i}} \Gamma_i)_{i\in\{1,2\}}$ *(from Figure 2) has the* $(\mathcal{E}, \mathcal{M})$*-factorization* $\Gamma_{\tau^{\sqcup}_\star} \xrightarrow{r} (\Gamma_\sqcup \xrightarrow{\theta^{\sqcup}_i} \Gamma_i)_{i\in\{1,2\}}$ *(as in Figure 3), then* $\exists \Gamma_\sqcup.\tau^{\sqcup}_G[r]$ *is an* $\mathcal{E}$*-tight existential supertype of both* $\exists \Gamma_1.\tau^1$ *and* $\exists \Gamma_2.\tau^2$*, and any other* $\mathcal{E}$*-tight existential supertype of both* $\exists \Gamma_1.\tau^1$ *and* $\exists \Gamma_2.\tau^2$ *is also an existential supertype of* $\exists \Gamma_\sqcup.\tau^{\sqcup}_G[r]$.

*Proof.* The existential type $\exists \Gamma_\sqcup.\tau^{\sqcup}_G[r]$ is $\mathcal{E}$-tight since $r$ belongs to $\mathcal{E}$ by definition of $(\mathcal{E}, \mathcal{M})$-factorization and $r$ equals $rep_{\tau^{\sqcup}_G[r]}$ by uniqueness of representers. By our theorem for existential subtyping, the morphism $\theta^{\sqcup}_1$ serves as evidence that $\exists \Gamma_\sqcup.\tau^{\sqcup}_G[r]$ is an existential supertype of $\exists \Gamma_1.\tau_1$, and likewise for $\theta^{\sqcup}_2$, because the appropriate diagrams commute by definition of $(\mathcal{E}, \mathcal{M})$-factorization.

The challenging part is proving that $\exists \Gamma_\sqcup.\tau^{\sqcup}_G[r]$ is the tight existential *join*. Suppose we have some other common tight existential supertype $\exists \Gamma.\tau$. By our earlier theorem we simply need to prove that $\tau^{\sqcup}_\star$ is a skeletal subtype of $\tau_\star$ and construct a morphism $\theta : \Gamma \to \Gamma_\sqcup$ making the appropriate diagram commute. By that same theorem we know $\tau^1_\star \leq_\star \tau_\star$ must hold and there must be a morphism $\theta_1$ such that the left-most square in the diagram below commutes, and likewise for $\theta_2$ (note that for clarity we drop the subscripts on the *gen* morphisms). Because $\tau^{\sqcup}_\star$ is assumed to be the skeletal join of $\tau^1_\star$ and $\tau^2_\star$, we know that $\tau^{\sqcup}_\star$ is a skeletal subtype of $\tau_\star$, so we have the central *gen* morphism in the diagram below. Furthermore, from uniqueness of *gen* morphisms we can deduce that both upper triangles in the diagram below commute. Lastly, the front squares in the diagram below commute by definition of $(\mathcal{E}, \mathcal{M})$-factorization. Thus the entire diagram below commutes.




We can rearrange this diagram into commuting square(s) similar to those we saw earlier when discussing factorization structures:




The top morphism $rep_\tau$ belongs to $\mathcal{E}$ because we assumed $\exists \Gamma.\tau$ is $\mathcal{E}$-tight. The bottom 2-source $(\Gamma_\sqcup \xrightarrow{\theta^{\sqcup}_i} \Gamma_i)$ belongs to $\mathcal{M}$ by definition of $(\mathcal{E}, \mathcal{M})$-factorization. Thus because $\mathbf{Bnd}$ has unique $(\mathcal{E}, \mathcal{M})$-diagonalizations by assumption, there is a unique commuting diagonal $\theta : \Gamma \to \Gamma_\sqcup$. By our earlier theorem, $\tau^{\sqcup}_G[r] \leq_{\Gamma_\sqcup} \tau[\theta]$ holds because $r$ equals $rep_{\tau^{\sqcup}_G[r]}$ and the appropriate diagram commutes since $\theta$ is a commuting diagonal for the above diagram. Therefore, as this holds for any such $\exists \Gamma.\tau$, $\exists \Gamma_\sqcup.\tau^{\sqcup}_G[r]$ is the $\mathcal{E}$-tight existential join of $\exists \Gamma_1.\tau^1$ and $\exists \Gamma_2.\tau^2$.

At last, we can use the construction in this theorem to define our abstract algorithm for joining tight existential types:

TIGHTJOINEXISTENTIALTYPES($\exists \Gamma_1.\tau^1$, $\exists \Gamma_2.\tau^2$):
  construct the skeletal join $\tau_\star^\sqcup$ of $\tau_\star^1$ and $\tau_\star^2$;
  construct the general context $\Gamma_{\tau_\star^\sqcup}$ and general body $\tau_G^\sqcup$ of $\tau_\star^\sqcup$;
  construct $gen_{\tau_\star^1 \leq_\star \tau_\star^\sqcup}$ and $gen_{\tau_\star^2 \leq_\star \tau_\star^\sqcup}$;
  construct $rep_{\tau^1}$ and $rep_{\tau^2}$;
  construct the maximally-constrained bound $\Gamma_\sqcup$ and $\mathcal{E}$-morphism $r$
    from the $(\mathcal{E}, \mathcal{M})$-factorization of $(gen_{\tau_\star^i \leq_\star \tau_\star^\sqcup}; rep_{\tau^i})_{i \in \{0,1\}}$;
    (the $\mathcal{M}$-source is not necessary)
  construct $\tau^\sqcup$ by applying substitution $[r]$ to $\tau_G^\sqcup$;
  return $\exists \Gamma_\sqcup.\tau^\sqcup$;

Note that this abstract algorithm reduces the problem of joining tight existential types to skeletal joins and $(\mathcal{E}, \mathcal{M})$-factorization. Skeletal joins tend to be easy to construct because all variables and constants have been removed from the problem. $(\mathcal{E}, \mathcal{M})$-factorization relies on domain-specific knowledge of the bounds being used. Hence, just like our abstract algorithm for existential subtyping, we have reduced the problem to skeletal types and domain-specific knowledge of bounds.

To understand the algorithm, consider the following two existentially quantified records (we use distinct bounds for clarity):

$$R_1 = \exists \alpha, \beta : \alpha \ll \beta. \langle \alpha, \alpha, \alpha, \beta \rangle$$
$$R_2 = \exists \delta, \varepsilon : \delta \ll \varepsilon. \langle \delta, \delta, \varepsilon, \varepsilon, \varepsilon \rangle$$

Their skeletal join and its general bound and body are as follows:

$$\tau_\star^\sqcup = \langle \star, \star, \star, \star \rangle$$
$$\Gamma_{\tau_\star^\sqcup} = a, b, c, d$$
$$\tau_G^\sqcup = \langle a, b, c, d \rangle$$

The compositions of their general subtyping morphisms and their representers are as follows:

| | $a$ | $b$ | $c$ | $d$ |
|---|---|---|---|---|
| $gen_{\tau_\star^1 \leq_\star \tau_\star^\sqcup}; rep_{\tau^1}$ | $\alpha$ | $\alpha$ | $\alpha$ | $\beta$ |
| $gen_{\tau_\star^2 \leq_\star \tau_\star^\sqcup}; rep_{\tau^2}$ | $\delta$ | $\delta$ | $\varepsilon$ | $\varepsilon$ |

Now we need to factor these two mappings using the (Surjection, Initial Mono-Source) factorization structure we defined earlier for our example $\mathbf{Bnd}$. We defined this factorization structure using the (Surjection, Mono-Source) factorization structure of $\mathbf{Set}$, which merges all variables that are mapped to the same value by both maps. In this case, $a$ and $b$ both map to $\alpha$ via the first map and to $\delta$ via the second map, so they are merged into a single variable. Thus $r$, the merging function, maps $a$ and $b$ to the same fresh variable $\nu$, $c$ to fresh variable $\pi$, and $d$ to fresh variable $\rho$. We can define the following mono-source in $\mathbf{Set}$ in terms of these fresh variables:

| | $\nu$ | $\pi$ | $\rho$ |
|---|---|---|---|
| $\theta_1^\sqcup$ | $\alpha$ | $\alpha$ | $\beta$ |
| $\theta_2^\sqcup$ | $\delta$ | $\varepsilon$ | $\varepsilon$ |

Next we need to turn the above mono-source in $\mathbf{Set}$ into an *initial* mono-source in $\mathbf{Bnd}$, so we need to constrain the fresh variables as much as possible such that the mappings $\theta_1^\sqcup$ and $\theta_2^\sqcup$ are still relation-preserving. Because inheritance is reflexive, $\nu$ is always mapped to a class which inherits the class mapped to by $\pi$, likewise for $\pi$ and $\rho$. Therefore we add the constraints $\nu \ll \pi$ and $\pi \ll \rho$. The variable $\nu$ is also always mapped to a class which inherits the class mapped to by $\rho$, but the constraint $\nu \ll \rho$ can be inferred from the other constraints since inheritance is transitive, so we leave this constraint implicit. Lastly, we simply put the pieces together and apply the substitution $[r]$ to the general type of the skeletal join in order to construct the tight existential join type:

$$\exists \nu, \rho, \pi : \nu \ll \pi \ll \rho. \langle \nu, \nu, \pi, \rho \rangle$$

In our experience [16], this abstract algorithm has been easy to implement and extend with more complex subtyping rules and more complex existential quantification. Now that we have an abstract algorithm for deciding existential subtyping and joining tight existential types, we have nearly all the pieces required by abstract interpretation. In order for abstract interpretation to terminate, we need our existentially quantified abstract domain to be co-well-founded and we need the dataflow function to be monotonic. Next we provide tools for both of these remaining requirements.

## 5 Abstract Interpretation

Abstract interpretation requires a co-well-founded abstract domain with decidable subtyping, joins, and a monotonic dataflow function [6]. We have already provided abstract algorithms for subtyping and joins. Here we provide tools for co-well-foundedness and for monotonicity. As we have seen, existential types can be surprisingly difficult, and we have used the requirements of our framework in order to reduce these difficulties down to skeletal types and domain-specific algorithms for bounds. We continue to do so here. We also provide techniques for optimally adding constraints deducible from branch guards in the language or control flow graph.

### 5.1 Co-Well-Foundedness

Co-well-foundedness for a preorder is the property that, given any infinite chain of supertypes, at some point that chain simply repeats the same element (up to isomorphism) forever. This property guarantees that the algorithm provided by abstract interpretation will always terminate (provided the dataflow function is monotonic, which we address next). Existential quantification makes the abstract domain much more expressive but also much more complex, so proving co-well-foundedness may seem daunting. However, by satisfying the requirements of our framework, co-well-foundedness of tight existential types reduces to co-well-foundedness of skeletal subtyping and a categorical property of $\mathbf{Bnd}$.

For a class of epimorphisms $\mathcal{E}$, each object $\mathcal{A}$ of a category has a preorder of $\mathcal{E}$-quotients [1]. An $\mathcal{E}$-quotient of $\mathcal{A}$ is a morphism $e : \mathcal{A} \to \mathcal{B}_e$ in $\mathcal{E}$ with domain $\mathcal{A}$ (the codomain can be any object). An $\mathcal{E}$-quotient of $\mathcal{A}$ is essentially $\mathcal{A}$ with additional constraints imposed upon it. The preordering of $\mathcal{E}$-quotients is defined by $e \leq e'$ holds whenever $e$' factors through $e$ (this is the opposite of the definition in [1]). Essentially, the ordering $e \leq e'$ holds whenever $e$ imposes fewer constraints than $e'$. If this ordering is well-founded it intuitively means that constraints cannot be strictly weakened ad infinitum. We use this to formalize an ordering of constraints on general bounds in our co-well-foundedness theorem.

**Theorem 3.** *Existential subtyping of $\mathcal{E}$-tight existential types in a system satisfying general subtyping is co-well-founded if skeletal subtyping is co-well-founded and the $\mathcal{E}$-quotient preorder for each general bound is well-founded.*

*Proof.* For simplicity, we assume skeletal subtyping is anti-symmetric. Suppose we have an infinite increasing sequence of $\mathcal{E}$-tight existential types $(\exists \Gamma_i.\tau_i)_{i \in \mathbb{N}}$ (with $\forall i \in \mathbb{N}.\, \exists \Gamma_i.\tau_i \sqsubseteq \exists \Gamma_{i+1}.\tau_{i+1}$). By our existential subtyping theorem, this implies that the skeletal types also form an infinite increase sequence. Since skeletal subtyping is co-well-founded, there must be some point $n$ at which the same skeletal type $\tau_\star$ is repeated forever. All existential types at $n$ and beyond then have the same general context $\Gamma_{\tau_\star}$ and so each can be classified by its representer $\mathit{rep}_{\tau_i}$. From uniqueness of $\mathit{gen}$ morphisms, we can deduce that $\mathit{gen}_{\tau_\star \leq_\star \tau_\star}$ is the identity morphism. Therefore our existential subtyping theorem informs us that, for $i$ larger than $n$, $\mathit{rep}_{\tau_i}$ factors through $\mathit{rep}_{\tau_{i+1}}$. By assumption each $\mathit{rep}_{\tau_i}$ belongs to $\mathcal{E}$, so for $i$ larger than $n$ we have that $\mathit{rep}_{\tau_{i+1}} \leq \mathit{rep}_{\tau_i}$ holds in the $\mathcal{E}$-quotient preorder for $\Gamma_{\tau_\star}$ by definition, and hence we have an infinite descending sequence. $\Gamma_{\tau_\star}$ is a general bound, so by assumption its $\mathcal{E}$-quotient preorder is well-founded. Thus, after some point $n'$ the $\mathit{rep}_{\tau_i}$ are all isomorphisms. Therefore, after $n'$ all existential types $\exists \Gamma_i.\tau_i$ are isomorphic. Since this holds for any infinite increasing sequence of existential types, tight existential subtyping is co-well-founded.

### 5.2 Monotonicity

Abstract interpretation requires all dataflow operations to be monotonic. This is, all dataflow operations must preserve the ordering on the abstract domain. Once gain, in an existentially quantified abstract domain the ordering can be quite complex. However, it turns out to be quite simple to extend a monotonic dataflow function over an abstract domain of simple types to a monotonic dataflow function over an abstract domain of existential types. The reason is that dataflow functions tend to be *natural*.

Consider the following Hoare triple [7]:

$$\{\texttt{EAX} \mapsto \textsc{Ptr}(\langle\!|\omega|\!\rangle)\} \quad \texttt{mov EAX [EAX]} \quad \{\texttt{EAX} \mapsto \textsc{Ins}(\omega)\}$$

If we were apply substitution with any mapping $\theta$ to both the precondition and the postcondition then the resulting Hoare triple would also hold. This means the dataflow function for the instruction `move EAX [EAX]` is *natural*, and dataflow functions typically fall in this category. In the appendices, we formalize this concept and formally state and prove the following useful theorem.

**Theorem 4.** *Any monotonic dataflow function over a simple abstract domain which is natural extends to a monotonic dataflow function over existentially quantified variants of that abstract domain. Furthermore if the original dataflow function is sound then so is the extension to the existentially quantified variants.*

### 5.3 Adding Constraints

A path-sensitive analysis needs to be able to gain information from branch guards. For example, a branch may check to see if two vtables are equal, and on the true branch the analysis should be able to use the fact that the corresponding classes are equal. Indeed, this is how our typed assembly language type checks runtime casts [16].

Existential types provide an excellent setting for adding constraints gained from path sensitivity. However, this process is not necessarily trivial. For example, suppose we have an existential type system for classes with inheritance which allows constraints of the form $\alpha \ll \beta$ but not $\texttt{Oval} \ll \beta$. In such a system, suppose we have an existential type such as $\exists\alpha, \beta : \alpha \ll \beta.\langle\alpha, \beta\rangle$ and we determine, say by comparing vtables, that in the current path $\alpha$ must equal $\texttt{Oval}$. Then if we choose to refine the type by replacing $\alpha$ with $\texttt{Oval}$ we would have to discard the inheritance constraint and forget that the first field inherits the second; however if we choose not to refine the type then we would not be able to exploit the fact that the first field is actually $\texttt{Oval}$. We say that such an existential type system is incompatible with equality inference.

We can formalize equality inference by using the categorical concept of *coequalizers* [1]. Coequalizers force specified values to become equal and *nothing more*; that is they do not add other constraints or make other values equal unless implied by the new equality. This is import for soundness, since an *unsound* way to resolve the dilemma above is by also making $\beta$ also equal $\texttt{Oval}$. An existential type system is compatible with equality inference if it has coequalizers and includes the empty bound, which we formalize next. Recalling our formalism for existentially quantified abstract domains from Section 2, an existential type abstracts all concrete types which arise from a valid substitution of its variables to concrete values. The empty bound must have the property that each valid mapping from a bound $\Gamma$ to constants corresponds to a permitted mapping from $\Gamma$ to the empty bound; thus the empty bound embodies constants. By having such an empty bound we ensure that coequalizers correspond to true equality in the concrete domain, ensuring soundness.

Coequalizers formalize equality inference, but there are many more constraints to be gained from path-sensitivity. We can use coinserters [9] to formalize inheritance constraints and integer inequalities. Specialized pushouts [1] can formalize the above kinds of constraints and more. All of these are known as *couniversal properties* [1]. By using couniversal properties to formalize adding constraints we ensure monotonicity, and in the presence of an empty bound we also ensure soundness.

## 6 Nested Existential Quantification

We have now specified all the requirements necessary for a complete inference algorithm of tight existential types. However, we have not addressed a common component of existential type systems: nested existential quantification. For example, in object-oriented languages fields are rarely exact instances of some statically known class but rather instances of some subclass of a statically known class. Furthermore, the specific subclass often changes as the program runs because the field keeps being assigned new instances of different subclasses of the statically known class. To express this, we might give the field a *nested* existential type, such as $\exists\alpha \ll \texttt{Oval}.\textsc{Ins}(\alpha)$.

Nested existential quantification is problematic, however, because nested existential subtyping does not fit within our framework. Consider the following rule for nested existential subtyping:

$$\frac{\omega \ll \omega' \text{ holds in the bound } \Gamma}{\exists\alpha \ll \omega.\textsc{Ins}(\alpha) \sqsubseteq_\Gamma \exists\alpha \ll \omega'.\textsc{Ins}(\alpha)}$$

The problem is that this rule cannot meet our requirement for general subtyping; it does not satisfy the rule of thumb for general subtyping that each metavariable in the supertype (in this case $\omega'$) should occur in the subtype.

In this section we describe techniques for navigating this problem. In our experience, these techniques have been able to address all our needs [16]. First, since we are not using full nested existential subtyping, for sake of clarity we also do not use the syntax for nested existential quantification. Instead we introduce *openable* types such as $\textsc{SubIns}(\omega)$ to represent the nested existential types which arise in the specific type system at hand. Although they do not use full nested existential subtyping, they are still permitted some subtyping provided it fits within our framework. For

example, we can use the subtyping rule $\textsc{Ins}(\omega) \leq \textsc{SubIns}(\omega)$. To properly handle these hidden existential quantifiers, we *always* open these types whenever possible, as we will describe first. To deal with our restricted subtyping, we use a special *assignability* relation when checking *requirements* such as checking that a value is assignable to a field, which we will describe second.

### 6.1 Opening

As mentioned before, traditional type systems with existential quantification use an **open** or **unpack** operation to pull out nested existential quantifiers [4, 5, 8, 11, 14]. We do the same with openable types, except we always do so implicitly whenever possible so that there is no need for explicit **open** operations. There is no loss of information in doing so; in fact it is best to open immediately. However, not every openable type in the body can be opened. For example, an openable type in a mutable field in memory cannot be opened because other operations may change the value of this field and consequently the existentially quantified values might change. However, an openable type in a local register can be opened, so we call this an *openable location*. Should an openable type occur in an openable location, we always open it implicitly, adding the quantified variables and constraints to the outer existential bound. The process of adding quantified variables and constraints can be formalized using pushouts [1], much like we discussed in the last section. This formalism also guarantees that opening is monotonic, as required by abstract interpretation.

### 6.2 Assignability

Unfortunately we cannot use nested existential subtyping in the ordering for an existentially quantified abstract domain for sake of complete analysis or inference. Rather, we must restrict existential subtyping to use restricted subtyping rules for openable types representing nested existential types. However, our strategy of always implicitly opening nested existential types as soon as possible seems to overcome this limitation. In our experience [16], we only needed nested existential subtyping when checking whether an operation was *valid*. For example, we needed nested existential subtyping when assigning a value of type $\textsc{Ins}(\alpha)$ to a field of type $\textsc{SubIns}(\texttt{Oval})$, for which we would simply check whether $\alpha$ inherits `Oval` under the current constraints. We also needed nested existential subtyping when checking whether the arguments to a function call were valid. Because of these usage patterns we defined a separate relation which we called *assignability*. Assignability is existential subtyping extended with nested existential quantification. However, assignability is not incorporated into the ordering on the existentially quantified abstract domain. Instead, assignability is used only for validity checks; that is assignability could only be used to determine if an operation, such as assigning a value to memory or calling a function, is valid. Although this need for distinguishing between subtyping and assignability is unfortunate, it is easy to implement and has been successful in our experience.

## 7 Conclusion

We have presented a framework for incorporating existential quantification into abstract interpretation. The framework applies a category-theoretic perspective on existential quantification in order to separate problems with existential quantification into separate and much simpler problems with skeletal types and a category of bounds and mappings. Using these techniques, the framework provides simple requirements which serve as useful guidelines while designing the existentially quantified abstract domain. Provided these requirements are satisfied, the framework provides abstract algorithms for deciding subtypes and for constructing joins, along with tools for proving co-well-foundedness, proving monotonicity, proving soundness, adding constraints, and handling nested existential quantifiers. The framework is described in bits and pieces throughout the paper, so we summarize it below.

We represent an existentially quantified abstract domain as a category $\mathbf{Bnd}$ of bounds and permitted variable mappings along with a functor $\langle Bodies, [-] \rangle : \mathbf{Bnd} \to \mathbf{Prost}$ specifying the bodies and subtyping for each bound as well as how to extend variable mappings to subtype-preserving substitutions. We define an existential type for such a system to have the form $\exists \Gamma.\tau$ where $\Gamma$ is an object of $\mathbf{Bnd}$ and $\tau$ is an element of $Bodies(\Gamma)$. We define existential subtyping with the following rule:

$$\frac{\theta : \Gamma' \to \Gamma \text{ in } \mathbf{Bnd} \qquad \tau \leq_\Gamma \tau'[\theta] \ (\leq_\Gamma \text{ specified by } Bodies(\Gamma))}{\exists \Gamma.\tau \sqsubseteq \exists \Gamma'.\tau'}$$

We then observe that every body $\tau$ in any bound $\Gamma$ has a skeletal type $\tau_\star$, a general bound $\Gamma_\tau$, a general body $\tau_G$ for that bound, and a representer $\mathit{rep}_\tau : \Gamma_\tau \to \Gamma$ with $\tau = \tau_G[\mathit{rep}_\tau]$.

Our framework has some requirements of the existentially quantified abstract domain. There must be a terminal object $\Gamma_\star$ in $\mathbf{Bnd}$. We call the preordered set $Bodies(\Gamma_\star)$ skeletal types with skeletal subtyping. We require a decision algorithm for skeletal subtyping and an algorithm for joining skeletal types. For each skeletal type $\tau_\star$ we require a general type $\exists\Gamma_{\tau_\star}.\tau_G$ with the property that, for any existential type $\exists\Gamma.\tau$ with skeletal type $\tau_\star$, there is a unique morphism $\mathit{rep}_\tau : \Gamma_{\tau_\star} \to \Gamma$ with $\tau = \tau_G[\mathit{rep}_\tau]$ called a representer. For each skeletal subtype $\tau_\star \leq_\star \tau'_\star$ we require a unique morphism $\mathit{gen}_{\tau_\star \leq_\star \tau'_\star} : \Gamma_{\tau'_\star} \to \Gamma_{\tau_\star}$ with $\tau_G \leq_{\Gamma_{\tau_\star}} \tau'_G[\mathit{gen}_{\tau_\star \leq_\star \tau'_\star}]$. We additionally require that for any subtype $\tau \leq_\Gamma \tau'$ in any bound $\Gamma$ we have the equality $\mathit{rep}_\tau \circ \mathit{gen}_{\tau_\star \leq_\star \tau'_\star} = \mathit{rep}_{\tau'}$. We require that $\mathbf{Bnd}$ have some chosen $(\mathcal{E}, \mathcal{M})$ factorization structure. We require a decision procedure for determining whether a morphism from any general bound factors through an $\mathcal{E}$ morphism from that same general bound. Lastly, we require an algorithm for constructing the intermediate object $\Gamma_\sqcup$ and $\mathcal{E}$ morphism $r$ portions of an $(\mathcal{E}, \mathcal{M})$-factorization of any 2-source.

We define a tight existential type to be an existential type whose representer belongs to $\mathcal{E}$. With the above requirements, we provide algorithms for deciding existential subtypes and for joining tight existential types. We also provide tools for proving tight existential subtyping to be well-founded and for the dataflow algorithm to be monotonic and sound. Should these hold as well, then using abstract interpretation we provide a complete sound inference algorithm or analysis for tight existential types.

All together these requirements seem overwhelming. However many of them are technical points formalizing intuitive type-theoretic concepts in category-theoretic terms, and many of them are extremely common with plenty of tools for satisfying them as shown in the appendices. In our experience [16], this framework was extremely instructive while designing an inferable type system with existential quantification, producing existentially quantified abstract domains which are very expressive, easy to extend, and flexible enough to adapt to new circumstances.

We are not the first to use existential subtyping though. As mentioned, Scala uses existential subtyping and consequently type checking is undecidable [17]. Java uses wildcards, essentially a restricted form of existential quantification, and type checking with wildcards is also believed to be undecidable [2, 10, 17]. Unfortunately, the impredicative nature of these type systems makes our framework unhelpful. In fact, we have tried applying it to Java wildcards and came across the same indicators of undecidability. However, in these explorations our framework has suggested ways to fix Java's types so that not only would it be decidable, but it could even use more expressive existential quantification that would also be inferable. We leave this investigation to future work however.

There have also been prior attempts at inference with existential quantification. One such attempt was with the existential types used internally in Pizza [15]; however, using the insight from the framework we actually identified a flaw in the join algorithm as described in the paper, which we have already discussed in prior work [16]. Coolaid[3] uses a dependent type system very similar to existential quantification. However, the join algorithm for this type system is incomplete because of the use of dependent types and integer arithmetic [3], and even its specialized handling of integer expressions would benefit from the same factorization structure we used in our inferable typed assembly language [16]. Thus, both prior type systems and prior inference systems with existential quantification could benefit from the use of our framework.

We have illustrated the potential of this framework by using it to design an inferable typed assembly language for C# [16] and demonstrated that it could also have contributed to prior systems and might still be able to contribute to existing systems. In our experience this framework is informative, powerful, flexible, and extensible. In particular, the abstraction granted by category theory allows our framework to adapt to the needs of the designer, rather than just the needs we anticipated. In fact, the framework is presented here with severe restriction for sake of brevity. In particular, one could actually use factorization structures for functors [1] to gain a whole new degree of freedom and ironically simplicity while designing an existentially quantified abstract domain. Our appendices provide much more thorough instructions of how to use our framework in practice, particularly in its generalized form.

That being said, there is still opportunity for improvement. The framework requires algorithms for deciding when one morphism factors through another and for constructing $(\mathcal{E}, \mathcal{M})$-factorizations. With these problems as motivations, we believe this opens a new line of both theoretical and applied research for constructing categories in which these problems are guaranteed to be computable. Nonetheless, we currently leave the domain-specific problems to the designer so that they may still benefit from our general framework for inference and analysis with existential quantification.

## References


1. J. Adámek, H. Herrlich, and G. Strecker. *Abstract and Concrete Categories*. Wiley-Interscience, New York, NY, USA, 1990.
2. Nicholas Cameron, Sophia Drossopoulou, and Erik Ernst. A model for Java with wildcards. In *ECOOP*, 2008.
3. B. E. Chang, A. Chlipala, G. C. Necula, and R. R. Schneck. Type-based verification of assembly language for compiler debugging. In *TLDI*, 2005.
4. J. Chen, C. Hawblitzel, F. Perry, M. Emmi, J. Condit, D. Coetzee, and P. Pratikaki. Type-preserving compilation for large-scale optimizing object-oriented compilers. In *PLDI*, 2008.
5. J. Chen and D. Tarditi. A simple typed intermediate language for object-oriented languages. In *POPL*, 2005.
6. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1977.
7. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.
8. B. Jacobs. *Categorical Logic and Type Theory*. Number 141 in Studies in Logic and the Foundations of Mathematics. North Holland, Amsterdam, 1999.
9. P. T. Johnstone. *Sketches of an Elephant: A Topos Theory Compendium*, volume 1. Oxford University Press, USA, October 2002.
10. Andrew Kennedy and Benjamin Pierce. On decidability of nominal subtyping with variance. In *FOOL*, 2007.
11. C. League, Z. Shao, and V. Trifonov. Type-preserving compilation of Featherweight Java. *TOPLAS*, 24(2):112–152, 2002.
12. G. Morrisett, K. Crary, N. Glew, D. Grossman, R. Samuels, F. Smith, D. Walker, S. Weirich, and S. Zdancewic. TALx86: A realistic typed assembly language. In *Workshop on Compiler Support for System Software*, 1999.
13. G. Morrisett, K. Crary, N. Glew, and D. Walker. Stack-based typed assembly language. *Journal of Functional Programming*, 13(5):957–959, 2003.
14. G. Morrisett, D. Walker, K. Crary, and N. Glew. From system F to typed assembly language. *TOPLAS*, 21(3):527–568, 1999.
15. M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In *POPL*, 1997.
16. Ross Tate, Juan Chen, and Chris Hawblitzel. Inferable object-oriented typed assembly language. In *PLDI*, 2010.
17. Stefan Wehr and Peter Thiemann. On the decidability of subtyping with bounded existential types. In *APLAS*, 2009.


## A Designing Inferable Bounds for Existential Quantification

In the appendices we demonstrate how to apply techniques from category theory in order to build various forms of existential quantification. We also show how to relax the requirements of our framework as described in the paper in order to enable yet more flexibility in the designs of these forms of existential quantification. We will begin in Section B by adding constants to the existential type system, an important ability required of most any existential quantification. This addition will demonstrate that the framework as described in the paper is overly restrictive, so in Section C we show how to relax the requirements of our paper by introducing a distinction between *bounds* and *contexts*. In Section D we will take advantage of this distinction in order to add various forms of algebraic structure, such as generics or arithmetic, to existential quantification. In Section E we present a technique for encoding axioms or inference rules in category theory so that we can identify which such rules will preserve the factorization structure required by the framework, and we summarize how to combine the techniques provided in the appendices in order to design inferable forms of existential quantification. Lastly in Section F we formalize the semantics of existential quantification and show how to easily extend sound monotonic natural dataflow functions over simple abstract domains to sound monotonic dataflow functions over existentially quantified abstract domains. We present these techniques and theorems so that analysis and type system designers may benefit from our experience with category theory and learn how to build inferable existentially quantified abstract domains.

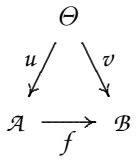
## B Constants

Constants are an important part of any existential type system because the quantified variables are meant to represent unknown constants. We will continue using the classes with inheritance example from the paper, but now we will add constant classes with predesignated inheritance. We will temporarily throw out our finiteness requirement so that we have an infinite set of constants, but we will regain this ability later. With this temporary relaxation, the category of classes with inheritance (without constants) is the category $\mathbf{Poset}$ of partially ordered sets.

### B.1 Categorical Interpretation

Now suppose we have our partially ordered set of constant classes with inheritance $\Theta$. $\Theta$ is itself an object of $\mathbf{Poset}$. Because of this, we can form what is known as the coslice category of $\Theta$ [8] or the category of objects under $\Theta$ [1], denoted as $\mathbf{Poset}\backslash\Theta$:

- Objects are pairs $\langle \mathcal{A}, u : \Theta \to \mathcal{A} \rangle$
- Morphisms from $\langle \mathcal{A}, u \rangle$ to $\langle \mathcal{B}, v \rangle$ are morphisms $f : \mathcal{A} \to \mathcal{B}$ such that the following commutes:




- Composition and identity are as in $\mathbf{Poset}$

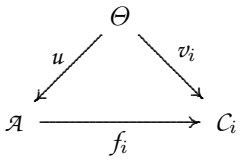
The intuition is that an object $\langle \mathcal{A}, u \rangle$ under $\Theta$ is a class hierarchy that contains $\Theta$ and $u$ specifies just how it contains $\Theta$ (i.e. $u$ is the inclusion function). Any element of $\mathcal{A}$ in the image of $u$ is a constant, and all elements not in the image of $u$ are variables. A morphism $f : \langle \mathcal{A}, u \rangle \to \langle \mathcal{B}, v \rangle$ is a relation-preserving function from $\mathcal{A}$ to $\mathcal{B}$ which preserves the contained $\Theta$ structure, that is $f$ is a function which preserves constants. Another way to think of out is that $f$ must map each element in the image of $u$ to the corresponding element in the image of $v$, but all other elements (i.e. the variables) can be mapped to either constants or variables in $\mathcal{B}$. Hence this formalism captures our basic intuition for constants.

Now we should consider full subcategories of this coslice category. A subcategory is a category formed from a subset of the objects and morphisms of a larger category. A *full* subcategory is formed from a subset of the objects but contains *all* morphisms between those objects. Thus by considering full subcategories of the coslice category we are simply selecting which objects $\langle \mathcal{A}, u \rangle$ under $\Theta$ we want to consider.

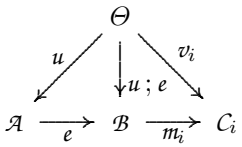
First, often with constants we know that these constants are distinct. The classes `Oval` and `Circle` will never represent the same class. However, so far there is nothing which prevents $u$ in $\langle \mathcal{A}, u \rangle$ from mapping two different constants to the same value in $\mathcal{A}$. Therefore in a setting with distinct constants we may want to permit only objects under $\Theta$ such that $u$ is a monomorphism (such as an injection in $\mathbf{Set}$ or $\mathbf{Rel}$). This way existential bounds cannot constrain two different constants to be equal. Similarly we may want to only consider existential bounds which are *satisfiable*; that is there should be a valid way to assign bound variables to constants. Formally, this requires that $u$ have at least one post-inverse: a morphism $sat : \mathcal{A} \to \Theta$ such that $u \,;\, sat$ equals $id_\Theta$. Morphisms which have a post-inverse are known as *sections* [1]. Thus we might consider using the full subcategory of sections under $\Theta$. However, satisfiability is a closed world concept; a bound which is not satisfiable now may be satisfiable in a future in which the programmer adds new modules extending the class hierarchy. Thus, we may want to at least require that an existential bound does not add new inheritance constraints between constants, such as `Square` $\ll$ `Oval`, which will never be true without actually changing the definitions of these constants. Interestingly this can be formalized by requiring $u$ to be an initial (mono)morphism. To summarize we have demonstrated that constants can be formalized with (some class of morphisms) under $\Theta$, where the class of morphisms depends on the specific application.

### B.2 Constants in our Framework

Here we consider the requirements of our framework in this setting with constants. For the coslice category, we can actually inherit the factorization structure of the original category. Suppose we have a source in our coslice category:



Notice that $(\mathcal{A} \xrightarrow{f_i} \mathcal{C}_i)_{i \in \mathcal{I}}$ forms a source in the original category, and so we can use the $(\mathcal{E}, \mathcal{M})$-factorizations of the original category to produce the following commutative diagram:




Thus the coslice category essentially inherits the factorizations of the original category. It can easily be shown that it also inherits the unique diagonalizations of the original category, and so inherits the entire factorization structure. Now let us reconsider our subcategories of certain morphisms under $\Theta$. These subcategories also inherit the entire factorization structure *provided that* knowing that $u$ and each $v_i$ are permitted then $u\,;e$ is also permitted in the above situation. In fact it can be shown that this holds for monomorphisms, sections, and $\mathcal{M}$ from the factorization structure using properties from [1]. In particular, it holds for initial monomorphisms in $\mathbf{Poset}$ because $\mathbf{Poset}$ has a (Surjection, Initial Mono-Source) factorization structure. Thus all the formalizations of constants that we described above inherit the factorization structure of the original category without constants.

We have just addressed factorization structures for constants, but recall that our framework also requires $\mathbf{Bnd}$ to have general bounds and a terminal object. General bounds can be resolved by using coproducts [1], but this is more cumbersome than it should be. Regardless, the various subcategories of the coslice category generally do not have a terminal object. Thus we cannot the formalisms we just described cannot be used by our framework *as described in the paper*, but in the next section we will show how to relax the requirements of our framework significantly. Our formalisms for constants easily satisfy these relaxed requirements and we will also be able to use a simpler formalism for general bounds.

# C Relaxing the Framework

In the paper we bundled two concepts for sake of simplicity: bounds and contexts. A context is used to determine which bodies are valid and how to do substitution. Each bound defines a context, but also defines more complex information and substitutions. For example, in many type systems a context simply defines the various class or integer terms which may be used as the value of a slot for bodies in the context. Contexts, on the other hand, also define inheritance or inequality constraints and have complex rules for substitution such as distinguishing variables in contexts. Here we show how to relax our framework by separating these two concepts into two separate categories $\mathbf{Cxt}$ and $\mathbf{Bnd}$ with a functor $Cxt : \mathbf{Bnd} \to \mathbf{Cxt}$ connecting the two concepts.

## C.1 Categorical Reinterpretation

First we reexamine our formalization of type systems and existential quantification. A type system is a category $\mathbf{Cxt}$ of contexts and mappings with a functor $\langle Types, [-] \rangle : \mathbf{Cxt} \to \mathbf{Prost}$ specifying the types and subtyping for each context and how to extend mappings of contexts to subtyping-preserving substitutions of types. A typing judgement $\Gamma \vdash \tau$ is valid when $\Gamma$ is an object of $\mathbf{Cxt}$ and $\tau$ is an element of $Types(\Gamma)$, and there is analogous formalism for subtyping judgements. An existential quantification for such a type system is a category $\mathbf{Bnd}$ of bounds and mappings and a functor $Cxt : \mathbf{Bnd} \to \mathbf{Cxt}$ specifying the context associated with each bound and how to convert mappings of bounds into mappings of contexts. The $\langle Bodies, [-] \rangle$ functor used in the paper is simply the composition of $Cxt$ and $\langle Types, [-] \rangle$. An existential type $\exists \Delta.\tau$ for such a system is an object $\Delta$ of $\mathbf{Bnd}$ and an element $\tau$ of $Types(Cxt(\Delta))$ (i.e. satisfying $Cxt(\Delta) \vdash \tau$). Existential subtyping then is defined by the following rule:

$$\frac{\theta : \Delta' \to \Delta \qquad Cxt(\Delta) \vdash \tau \leq \tau'[\theta]}{\exists \Delta.\tau \sqsubseteq \exists \Delta'.\tau'}$$

Before we proceed with the requirements of our framework, let us examine how our formalism for constants can be fit within this new perspective. Suppose we already have a type system without constants: $\langle Types, [-] \rangle : \mathbf{Cxt} \to \mathbf{Prost}$. Suppose then that our constants form an object $\Theta$ of the category $\mathbf{Cxt}$. We can then define $\mathbf{Bnd}$ as the coslice category (or a subcategory thereof) $\mathbf{Cxt} \backslash \Theta$. The functor $Cxt : \mathbf{Bnd} \backslash \Theta$ simply maps an object under $\Theta$, $\langle \mathcal{A}, u \rangle$, to the

object $\mathcal{A}$ in $\mathbf{Cxt}$; consequently the operation on morphisms is the obvious one. In our existential types perspective, this amounts to mapping the bound $\{\alpha, \beta\}$ to the context containing $\alpha$, $\beta$, and all constants. Because $\mathbf{Bnd}$ only has context-preserving morphisms, any substitution using a mapping of bounds will never replace constants with variables (although it may replace variables with constants as we would expect). Thus we have used our separation of bounds and contexts to separate constants and constant-preservation from types and simple substitution.

Now we adjust the requirements of our framework to this new formalism. First, we require $\mathbf{Cxt}$, rather than $\mathbf{Bnd}$, to have a terminal object $\Gamma_\star$. Types and subtyping in this context are the skeletal types and skeletal subtyping. This addresses our earlier problem with constants, since even though the category of monomorphisms under $\Theta$ (in $\mathbf{Prost}$) representing $\mathbf{Bnd}$ typically does not have a terminal object, the category $\mathbf{Prost}$ representing $\mathbf{Cxt}$ does have a terminal object as we described in the paper. Second, we require each skeletal type to have a general *context* in $\mathbf{Cxt}$ with the same unique representer requirement for any type in any *context* with that skeletal type. Similarly we require each skeletal type to have a unique *gen* morphism between general contexts satisfying the same equalities with representers of subtypes in any context. So far we have simply replaced $\mathbf{Bnd}$ with $\mathbf{Cxt}$, but our third requirement requires a factorization structure on the *functor* $Cxt : \mathbf{Bnd} \to \mathbf{Cxt}$. This is where the design of $\mathbf{Bnd}$ becomes key to designing expressive and efficient forms of existential quantification.

### C.2 Factorization Structures on Functors

Before discussing factorizations structures on functors we have to discuss *structured sources*, and for that we will first introduce categories of algebras: $\mathbf{Alg}(\Omega)$. $\Omega$ is a set of operators and an arity (possibly infinite) for each operator. For simplicity we will consider the category $\mathbf{Alg}(2)$ of sets equipped with a binary operator $bop$:

- Objects $\mathcal{A}$ are pairs $\langle \mathcal{A}, bop_{\mathcal{A}} \rangle : \mathcal{A} \times \mathcal{A} \to \mathcal{A}$
- Morphisms $f : \langle \mathcal{A}, bop_{\mathcal{A}} \rangle \to \langle \mathcal{B}, bop_{\mathcal{B}} \rangle$ are all functions $f : \mathcal{A} \to \mathcal{B}$ making the following diagram commute:

$$\begin{array}{ccc} \mathcal{A} \times \mathcal{A} & \xrightarrow{f \times f} & \mathcal{B} \times \mathcal{B} \\ {\scriptstyle bop_{\mathcal{A}}}\downarrow & & \downarrow{\scriptstyle bop_{\mathcal{B}}} \\ \mathcal{A} & \xrightarrow[f]{} & \mathcal{B} \end{array}$$

- Composition and identity are that of functions

A morphism as defined above is known as an algebra homomorphism. Requiring the diagram to commute means that the equality $f(a\ bop_{\mathcal{A}}\ a') = f(a)\ bop_{\mathcal{B}}\ f(a')$ must hold for all $a$ in $\mathcal{A}$; that is $f$ preserves the algebraic structure or distributes through the binary operator. For example, multiplication by any constant integer is an algebra homomorphism from $\langle \mathbb{Z}, + \rangle$ to itself since multiplication distributes through addition. This category has a functor $U : \mathbf{Alg}(2) \to \mathbf{Set}$ simply mapping each algebra $\langle \mathcal{A}, bop_{\mathcal{A}} \rangle$ to its underlying set $\mathcal{A}$, with the obvious mapping on morphisms.

Now we move on to *structured morphisms*, using $\mathbf{Alg}(\Omega)$ as our running example. An $F$-structured morphism is a morphism of the form $\mathcal{A} \xrightarrow{f} F(\mathcal{B})$ [1]. Using $U$ from the above example, a $U$-structured morphism is a function $f$ from a set $\mathcal{X}$ to the underlying set $\mathcal{A}$ of an algebra $\langle \mathcal{A}, bop_{\mathcal{A}} \rangle$. An $F$-structured morphism $g : \mathcal{A} \to F(\mathcal{B})$ is said to be *generating* [1] if it satisfies the following:

$$\forall m, n : \mathcal{B} \to \mathcal{C}.\ \ g\,;F(m) = g\,;F(n) \implies m = n$$

This property is essentially the $F$-structured analog to epimorphisms (i.e. surjections). For example, in our category $\mathbf{Alg}(2)$, the inclusion function $\{1\} \to U(\langle \{n \in \mathbb{Z} \mid n > 0\}, + \rangle)$ is generating because all positive integers can be formed by expressions using only the constant 1 and the algebraic operator $+$. However, the inclusion function $\{1\} \to U(\langle \mathbb{Z}, + \rangle)$ is *not* generating since 0 and negative integers cannot be formed in such a way, but $\{-1, 1\} \to U(\langle \mathbb{Z}, + \rangle)$ is generating. Also, the inclusion function $\{1\} \to U(\langle \{n \in \mathbb{Z} \mid n > 0\}, * \rangle)$ is *not* generating because only 1 can be expressed using only 1 and $*$; thus the specific binary operator, as well as the underlying set, is important to recognize.

The functor $U : \mathbf{Alg}(2) \to \mathbf{Set}$ has the interesting property that any $U$-structured morphism can be factored into a generating function and an algebra monomorphism. Given a $U$-structured function $f : \mathcal{X} \to U(\langle \mathcal{A}, bop_{\mathcal{A}} \rangle)$, form the subset $\mathcal{A}_f$ of $\mathcal{A}$ expressible with just the elements $f(x)$ and the operator $bop_{\mathcal{A}}$. This subset is by construction

closed under $bop_{\mathcal{A}}$, so it forms a binary algebra $\langle \mathcal{A}_f, bop_f \rangle$. Let $g : \mathcal{X} \to U(\langle \mathcal{A}_f, bop_f \rangle)$ be the $U$-structured function mapping each element $x$ to $f(x)$ in $\mathcal{A}_f$. $g$ is generating by construction. Let $m : \mathcal{A}_f \to \mathcal{A}$ be the injective function including the subset $\mathcal{A}_f$ into $\mathcal{A}$. By definition of $bop_f$, $m$ always satisfies the equality $m(a\ bop_f\ a') = m(a)\ bop_{\mathcal{A}}\ m(a')$ and so we have an algebra monomorphism $\mathit{m} : \langle \mathcal{A}_f, bop_f \rangle \to \langle \mathcal{A}, bop_{\mathcal{A}} \rangle$. Furthermore, $g\,;U(\mathit{m})$ equals $f$, so we have factored $f$ into a generating function and an algebra monomorphism.

The functor $U : \mathbf{Alg}(2) \to \mathbf{Set}$ also has an interesting unique-diagonalization property. Given the following commutative diagram

$$\begin{array}{ccc} \mathcal{X} & \xrightarrow{\quad g \quad} & U(\langle \mathcal{A}, bop_{\mathcal{A}} \rangle) \\ {\scriptstyle u}\downarrow & & \downarrow{\scriptstyle U(\mathit{v})} \\ U(\langle \mathcal{B}, bop_{\mathcal{B}} \rangle) & \xrightarrow[U(\mathit{m})]{} & U(\langle \mathcal{C}, bop_C \rangle) \end{array}$$

where $g$ is a generating function and $\mathit{m}$ is an algebra monomorphism, there always exists a unique commuting diagonal, specifically a unique algebra homomorphism $\mathit{d} : \langle \mathcal{A}, bop_{\mathcal{A}} \rangle \to \langle \mathcal{B}, bop_{\mathcal{B}} \rangle$ with $g\,;U(\mathit{d}) = u$ and $\mathit{d}\,;\mathit{m} = \mathit{v}$. $d$ is easy to define:

$$d(g(x)) = u(x)$$
$$d(a\ bop_{\mathcal{A}}\ a') = d(a)\ bop_{\mathcal{B}}\ d(a')$$

It is simple to prove that this is a well-defined function by our assumptions that $g\,;U(v)$ equals $u\,;U(\mathit{m})$, $g$ is a generating function, and $\mathit{m}$ is an algebra monomorphism [1]. Furthermore $d$ satisfies the equality $d(a\ bop_{\mathcal{A}}\ a') = d(a)\ bop_{\mathcal{B}}\ d(a')$ by construction and so forms an algebra monomorphism $\mathit{d} : \langle \mathcal{A}, bop_{\mathcal{A}} \rangle \to \langle \mathcal{B}, bop_{\mathcal{B}} \rangle$. The remaining requirements are easy to prove from the definition of $\mathit{d}$.

These two properties can also be shown to hold for their more general *source* counterparts, and so $U$ is said to have a (Generating, Mono-Source) factorization structure. It should be easy to see how this generalizes to $(\mathcal{G}, \mathcal{M})$ factorization structures for $F : \mathbf{C} \to \mathbf{D}$ where $\mathcal{G}$ is some class of $F$-structured morphisms (typically generating) and $\mathcal{M}$ is some class of sources in $\mathbf{C}$ (typically some kind of mono-source). In fact, a $(\mathcal{E}, \mathcal{M})$ factorization structure of a category $\mathbf{C}$ can be seen as actually a factorization structure for the identify functor $Id_{\mathbf{C}} : \mathbf{C} \to \mathbf{C}$, so factorization structures for functors generalize factorization structures for categories.

### C.3 Joining Existential Types

Now we can state our last requirement: $Cxt : \mathbf{Bnd} \to \mathbf{Cxt}$ must have some chosen $(\mathcal{G}, \mathcal{M})$ factorization structure. We say an existential type $\exists \Delta.\tau$ is $\mathcal{G}$-tight if the $Cxt$-structured morphism $\mathit{rep}_\tau : \Gamma_\tau \to Cxt(\Delta)$ belongs to $\mathcal{G}$.

**Theorem 5.** *If $Cxt : \mathbf{Bnd} \to \mathbf{Cxt}$ has a $(\mathcal{G}, \mathcal{M})$ factorization structure, then given two existential types $\exists \Delta_1.\tau^1$ and $\exists \Delta_2.\tau^2$, suppose $\tau^{\sqcup}_\star$ is the join of the skeletal types $\tau^1_\star$ and $\tau^2_\star$ with respect to skeletal subtyping, and the $Cxt$-structured 2-source $(\Gamma_{\tau^{\sqcup}_\star} \xrightarrow{\mathit{gen}_{\tau^i_\star \le \star \tau^{\sqcup}_\star}} \Gamma_{\tau^i} \xrightarrow{\mathit{rep}_{\tau^i}} Cxt(\Delta_i))_{i \in \{1,2\}}$ has the $(\mathcal{G}, \mathcal{M})$-factorization $\Gamma_{\tau^{\sqcup}_\star} \xrightarrow{r} Cxt(\Delta_\sqcup \xrightarrow{\theta^{\sqcup}_i} \Delta_i)_{i \in \{1,2\}}$, then $\exists \Delta_\sqcup.\tau^{\sqcup}_G[r]$ is a $\mathcal{G}$-tight existential supertype of both $\exists \Delta_1.\tau^1$ and $\exists \Delta_2.\tau^2$, and any other $\mathcal{G}$-tight existential supertype of both $\exists \Delta_1.\tau^1$ and $\exists \Delta_2.\tau^2$ is also an existential supertype of $\exists \Delta_\sqcup.\tau^{\sqcup}_G[r]$.*

The proof and abstract algorithm are essentially the same as in the paper.

### C.4 Restricting Constraints

Here we give an interesting application of this new degree of flexibility which we found useful for making our handling of integer inequality constraints efficient. Unfortunately we will not describe our techniques formally because doing so would require a digression into concrete-category theory [1]. However, we believe the reader should still be able to adapt the informal strategy we describe to their own domain.

In one of our earlier versions of our inferable typed assembly language [16], we used the optimally precise factorization structure of our category $\mathbf{Bnd}$ for our integer inequality constraints. This has the effect of preserving *all* constraints possible when joining existential types. The problem was, although this usually worked well, occasionally some fluke of coincidence would cause a quadratic blow-up of our integer inequality constraints. Because integer arithmetic is complex, simply propagating these constraints through substitutions was rather difficult. These problems added up, and we found inference time was being prolonged due to these constraints. Upon examining the constraints,

we identified that most of them were useless. The constraints that were useful were always the constraints on just the simple integer expressions *present in the body*, so we decided to try changing our factorization structure on the category $\mathbf{Bnd}$ of integer inequality constraints to a factorization structure on the functor $Cxt$ mapping a bound to the set of simple integer expressions valid in that bound.

The observation we made was that each morphism $rep_\tau$ would simply specify the simple integer expression assigned to each slot in $\tau$. Thus, by requiring that all (explicit) constraints be restricted to just the image of the representer, we effectively restrict all explicit constraints to be just on the simple integer expressions assigned to slots in the body. This also has the benefit that the number of constraints in a $\mathcal{G}$-tight existential type is limited by the number of slots in the body, which in our typed assembly language only gets smaller so we do not have to worry about any massive increases of the number of constraints in a bound. We can define the class of $Cxt$-structured morphisms $\mathcal{G}$ to be all generating functions $g : \mathcal{X} \to Cxt(\mathcal{A})$ with the property that all explicit constraints in $\mathcal{A}$ are on only elements in the image of $g$. The problem is there is no class $\mathcal{M}$ of sources in $\mathbf{Bnd}$ which forms a factorization structure with $\mathcal{G}$. Rather, we have to generalize our framework just a bit to make $\mathcal{M}$ *dependent*.

A $Cxt$-dependent source class $\mathcal{M}$ specifies, for each $Cxt$-structured morphism $f : \mathcal{X} \to Cxt(\mathcal{A})$, a class of sources $\mathcal{M}(f)$ in $\mathbf{Bnd}$ with domain $\mathcal{A}$. $\mathcal{M}$ must also satisfy the property that $\mathcal{M}(f)$ is always a subset of $\mathcal{M}(g\,;f)$ for any $g : \mathcal{Y} \to \mathcal{X}$. For such a dependent $\mathcal{M}$, a $(\mathcal{G}, \mathcal{M})$ factorization of a $Cxt$-structured source $(\mathcal{X} \xrightarrow{f_i} Cxt(\mathcal{A}_i))_{i\in\mathcal{I}}$ is a factorization $\mathcal{X} \xrightarrow{g} Cxt(\mathcal{A} \xrightarrow{m_i} \mathcal{A}_i)_{i\in\mathcal{I}}$ such that $g$ is in $\mathcal{G}$ and $(\mathcal{A} \xrightarrow{m_i} \mathcal{A}_i)_{i\in\mathcal{I}}$ is in $\mathcal{M}(g)$. The unique $(\mathcal{G}, \mathcal{M})$-diagonalization property is that for any set of commutative diagram such as the following, where $g$ belongs to $\mathcal{G}$ and the source $(\mathcal{B} \xrightarrow{m_i} \mathcal{C}_i)$ belongs to $\mathcal{M}(u)$, there is a unique commuting diagonal $d : \mathcal{A} \to \mathcal{B}$:

$$\begin{array}{ccc} \mathcal{X} & \xrightarrow{g} & Cxt(\mathcal{A}) \\ {\scriptstyle u}\downarrow & & \downarrow{\scriptstyle Cxt(v_i)} \\ Cxt(\mathcal{B}) & \xrightarrow[Cxt(m_i)]{} & Cxt(\mathcal{C}_i) \end{array}$$

Note that, even with these relaxations, the proof for joining existential types still works, and so our framework can use these relaxed requirements instead.

In our example, $\mathcal{M}(f)$ is all mono-sources $(\mathcal{A} \xrightarrow{m_i} \mathcal{B}_i)_{i\in\mathcal{I}}$ such that the constraints are as strong as possible on the image of $f$ for each $m_i$ to be relation-preserving (i.e. initial on the image of $f$). It is easy to show that $\mathcal{M}(f)$ is a subset of $\mathcal{M}(g\,;f)$ for any $g$ since the image of $g\,;f$ is always smaller than the image of $f$. For $(\mathcal{G}, \mathcal{M})$-factorizations, we use the standard techniques for building functions $g$ and each $m_i$; then, for each pair of values $a$ and $a'$ in the image of $g$, we check whether the constraint $m_i(a) < m_i(a')$ holds in each target context, and if so we add the constraint $a < a'$. This makes implementing $(\mathcal{G}, \mathcal{M})$ factorizations *much* easier especially should we use complex integer constraint solvers. For unique diagonalizations, we use the standard techniques for showing a well-defined diagonal $d$, and this function is relation-preserving because $\mathcal{G}$ requires all constraints to be confined to the image and $\mathcal{M}$ requires all constraints to be as strong as possible on the image. Thus, although $Cxt$ has a more precise factorization structure, this $(\mathcal{G}, \mathcal{M})$ is much more efficient and easier to implement, and the framework grants us the flexibility to decide which system we prefer.

## D Injective Algebras

In the last section we presented categories of algebras $\mathbf{Alg}(\Omega)$. Algebraic structure in existential quantification takes the form of terms. For example, when using a generic class such as `List<T>` we can form terms for classes in a context with class variable $\alpha$: $\alpha$, `List<`$\alpha$`>`, `List<List<`$\alpha$`>>`, .... Thus, the set of classes forms an algebra with a unary operator `List`. Simple integer expressions form an algebra too: the terms $c_1 + i * c_2$ form a set with operators $(c+)$ and $(*c)$. However, we have to recognize an additional special property of these algebras in order to use them properly.

Consider the existential type $\exists\alpha.\langle$`List<`$\alpha$`>`$\rangle$. Using the (Generating, Mono-Source) factorization structure for $U$, this existential type is *not* tight. The general type is $x \vdash \langle x \rangle$ and the representer is $\{x \mapsto$ `List<`$\alpha$`>`$\}$. Notice that $\alpha$ itself cannot be expressed using only the value `List<`$\alpha$`>` and the operator `List`; thus the representer for this type is not generating. However, we need to be able to use the above type in order to type check polymorphic uses of generic

classes. Thankfully we can because generics actually form an *injective* algebra or *mono*-algebra, so we can use the subcategory $\mathbf{MonoAlg}(\Omega)$, in which all algebraic operators are injective, instead of $\mathbf{Alg}(\Omega)$.

Let $U' : \mathbf{MonoAlg}(\Omega) \to \mathbf{Set}$ be the functor mapping each mono-algebra to its underlying set. Suppose we have a mono-algebra $\mathcal{A}$, which can be seen as either an object of $\mathbf{Alg}(\Omega)$ or an object of $\mathbf{MonoAlg}(\Omega)$. In particular, any function $f : \mathcal{X} \to U(\mathcal{A}) = U'(\mathcal{A})$ is both a $U$-structured morphism and a $U'$-structured morphism. If $f$ is $U$-generating, then $f$ is also $U'$-generating. However, the opposite is not true. For example, $\{x \mapsto \texttt{List<}\alpha\texttt{>}\}$ is $U'$-generating but not $U$-generating. The reason is that a $U'$-generating function is only generating for morphisms to mono-algebras, and may not be generating for morphisms to all algebras. Suppose there were two algebra homomorphisms $f$ and $g$ from terms generated by $\alpha$ and `List` to some other mono-algebra $\mathcal{A}$ with the property that $\{x \mapsto \texttt{List<}\alpha\texttt{>}\} ; f$ equals $\{x \mapsto \texttt{List<}\alpha\texttt{>}\} ; g$. In particular, this means that $f(\texttt{List<}\alpha\texttt{>})$ equals $g(\texttt{List<}\alpha\texttt{>})$. Because both are algebra homomorphisms, this implies that $\texttt{List<}f(\alpha)\texttt{>}$ equals $\texttt{List<}g(\alpha)\texttt{>}$ (where `List` is also used to represent the unary operator in $\mathcal{A}$). Because $\mathcal{A}$ is a mono-algebra `List` is injective, so this implies that $f(\alpha)$ equals $g(\alpha)$. Because the algebra of terms generated by $\alpha$ and `List` is generated by $\alpha$, this implies that $f$ equals $g$. Since this holds for any such $f$ and $g$ to mono-algebras, $\{x \mapsto \texttt{List<}\alpha\texttt{>}\}$ is $U'$-generating. Note that the proof relied on the fact that $\mathcal{A}$ is a mono-algebra, and it would not work for algebras in general.

Now that we have established this distinction, $U'$ also has a (Generating, Mono-Source) factorization structure, where generating means $U'$-generating. Since $U'$-generating includes $U$-generating, this factorization structure produces a more precise tight existential type system. For example, $\exists\alpha.\langle\texttt{List<}\alpha\texttt{>}\rangle$ *is* a tight existential type with respect to this new factorization structure. When joining two types, if a slot $x$ is mapped to $\texttt{List<}\omega\texttt{>}$ in one body and $\texttt{List<}\omega'\texttt{>}$ in the other body, then $x$ will be mapped to a List of something in the join body, and if $\omega$ and $\omega'$ are themselves both a List of something then the process repeats recursively. Thus the join of existential types $\exists\alpha.\langle\texttt{List<List<List<}\alpha\texttt{>>>}\rangle$ and $\exists\beta.\langle\texttt{List<List<}\beta\texttt{>>}\rangle$ using this factorization structure is $\exists\gamma.\langle\texttt{List<List<}\gamma\texttt{>>}\rangle$, as we would expect when using generics.

Note that these same techniques work for *simple* integer expressions. $(c+)$ is injective for any constant $c$. $(*c)$ is injective for any non-zero constant $c$ (using non-modular arithmetic). The fact that $(*0)$ is not injective means that $\exists i.\{\texttt{EAX} \mapsto \textsc{Int}(i * 0)\}$ is not a tight existential type, which makes sense since it is equivalent to $\exists i.\{\texttt{EAX} \mapsto \textsc{Int}(0)\}$ in which $i$ does not occur in the body. The fact that $(*c)$ is not injective for modular arithmetic is why we do not consider $\exists\varnothing.\{\texttt{EAX} \mapsto \textsc{Int}(0)\}$ and $\exists\varnothing.\{\texttt{EAX} \mapsto \textsc{Int}(2^{32})\}$ to be equivalent types in our inferable typed assembly language [16]. Also, even though $(c+)$ is injective for any constant $c$, $+$ is not itself injective, likewise for $*$, which is why we restrict ourselves to *simple* integer expressions and do not allow expressions such as $i + j$. Having these guidelines from our framework to help us identify all these subtleties is one reason why the framework was so useful to have available while designing our inferable typed assembly language.

# E Axioms and Inference Rules

We have shown how to encode constraints in bounds by using a category of relations, and how to encode terms and expressions by using a category of algebras; here we show how to categorically encode axioms and inference rules such as reflexivity, transitivity, anti-symmetry, $0 * x = 0$, $1 * x = x$, multiplication distributes through addition, every class inherits `Object`, covariance of arrays, and even that any class which extends an array must itself by an array. Before we do this we will present a general-purpose category that combines algebras and relations and which will serve as a basis for these axioms and inference rules.

## E.1 $\mathbf{Mono}(\mho)\mathbf{PAlg}(\Omega)\mathbf{Rel}(\Phi)$

$\mathbf{Mono}(\mho)\mathbf{PAlg}(\Omega)\mathbf{Rel}(\Phi)$ is the category of partial algebras, with some injective operators, and relations. $\Omega$ is a set of operators and an arity (possibly infinite) for each operator. $\mho$ specifies a subset of operators in $\Omega$ which must be injective. $\Phi$ is a set of relations and an arity (possibly infinite) for each relation. An object $\mathcal{A}$ of this category is a set $\mathcal{A}$ with

- a partial function $op^{\omega}_{\mathcal{A}} : \mathcal{A}^{arity(\omega)} \rightharpoonup \mathcal{A}$ for each operator $\omega$ in $\Omega$
- such that $op^{\omega}_{\mathcal{A}}$ is injective if $\omega$ is in $\mho$,
- and a subset $\phi_{\mathcal{A}}$ of $\mathcal{A}^{arity(\phi)}$ for each relation $\phi$ in $\Phi$.

A morphism $f : \mathcal{A} \to \mathcal{B}$ in this category is a function $f : \mathcal{A} \to \mathcal{B}$ satisfying

- for each operator $\omega$ in $\Omega$, if $op^{\omega}_{\mathcal{A}}$ is defined on $\boldsymbol{a}$ then $op^{\omega}_{\mathcal{B}}$ is defined on $\boldsymbol{f(a)}$ and furthermore $f(op^{\omega}_{\mathcal{A}}(\boldsymbol{a}))$ equals $op^{\omega}_{\mathcal{B}}(\boldsymbol{f(a)})$
- and for each relation $\phi$ in $\Phi$, if $\boldsymbol{a}$ is in $\phi_{\mathcal{A}}$ then $\boldsymbol{f(a)}$ is in $\phi_{\mathcal{B}}$.

Thus morphisms are simply functions which preserve both the algebraic and relational structure of the objects.

We can make a sorted version of this category; that is one where each object has many underlying sets and the operators map elements from one underlying set to another and so on. However, a sorting defines an object $\Xi$ of $\mathbf{Mono}(\mho)\mathbf{PAlg}(\Omega)\mathbf{Rel}(\Phi)$. The category of objects *over* $\Xi$ [1] is actually equivalent (essentially isomorphic) to the sorted version of $\mathbf{Mono}(\mho)\mathbf{PAlg}(\Omega)\mathbf{Rel}(\Phi)$ using $\Xi$. Recall that the category of objects under $\Theta$ inherits factorization structures from the original category, and likewise the category of objects over $\Xi$ inherits factorization structures from $\mathbf{Mono}(\mho)\mathbf{PAlg}(\Omega)\mathbf{Rel}(\Phi)$. So we will focus on $\mathbf{Mono}(\mho)\mathbf{PAlg}(\Omega)\mathbf{Rel}(\Phi)$ and rely on this inheritance mechanism to transfer the results we find to the sorted version as well.

$\mathbf{Mono}(\mho)\mathbf{PAlg}(\Omega)\mathbf{Rel}(\Phi)$ has many useful factorization structures but we will focus on the most precise one: the (Epimorphism, Initial Mono-Source) factorization structure. A morphism $\mathcal{A} \xrightarrow{f} \mathcal{B}$ is an epimorphism if all elements of $\mathcal{B}$ belong to the smallest set

- containing the image of $f$,
- closed under each operator $\omega$ in $\Omega$ when defined in $\mathcal{B}$,
- and containing $op^{\omega}_{\mathcal{B}}(\boldsymbol{b})$ for any $\omega$ in $\Omega$ only if it also contains every $b$ in $\boldsymbol{b}$.

A source $(\mathcal{A} \xrightarrow{f_i} \mathcal{B}_i)_{i \in \mathcal{I}}$ is an initial mono-source whenever

- $(\mathcal{A} \xrightarrow{f_i} \mathcal{B}_i)_{i \in \mathcal{I}}$ is a mono-source in $\mathbf{Set}$,
- for each operator $\omega$ in $\Omega$, $op^{\omega}_{\mathcal{A}}$ is defined on $\boldsymbol{a}$ whenever $op^{\omega}_{\mathcal{B}_i}$ is defined on $\boldsymbol{f_i(a)}$ for all $i$ in $\mathcal{I}$,
- and for each relation $\phi$ in $Phi$, $\boldsymbol{a}$ is in $\phi_{\mathcal{A}}$ whenever $\boldsymbol{f_i(a)}$ is in $\phi_{\mathcal{B}_i}$ for all $i$ in $\mathcal{I}$.

Thus an (Epimorphism, Initial Mono-Source)-factorization produces the largest partial algebra generated by the source which is as constrained as possible. The proof that there are always unique (Epimorphism, Initial Mono-Source)-factorizations essentially combines the analogous proofs for $\mathbf{Rel}$ and $\mathbf{Alg}(\Omega)$. Thus we have a category which combines algebras and relations and has the most precise factorization structure possible.

### E.2 Categorical Implications

Many axioms can be encoded using *categorical implications* [1]. For example, we can encode symmetry as the following morphism in $\mathbf{Rel}(\approx\colon 2)$:

$$\mathit{sym} : \langle \{x, y\}, \{x \approx y\} \rangle \xrightarrow{\{x \mapsto x, y \mapsto y\}} \langle \{x, y\}, \{x \approx y, y \approx x\} \rangle$$

In the domain we have the variables $x$ and $y$ with the constraint $x \approx y$, and on the right we have added the constraint $y \approx x$. Essentially this corresponds to the implication $\forall x, y.\ x \approx y \Rightarrow y \approx x$. An object $\mathcal{R}$ of $\mathbf{Rel}(\approx\colon 2)$ is said to satisfy the categorical implication *sym* if every morphism $f : \langle \{x, y\}, \{x \approx y\} \rangle \to \mathcal{R}$ factors through *sym*; that is, there is a morphism $f' : \langle \{x, y\}, \{x \approx y, y \approx x\} \rangle \to \mathcal{R}$ such that $\mathit{sym}\,; f'$ equals $f$. Such a morphism $f$ picks out two elements $f(x)$ and $f(y)$ in $\mathcal{R}$ such that $f(x) \approx_{\mathcal{R}} f(y)$ holds. Since *sym* essentially just adds the constraint $y \approx x$, $f$ factors through *sym* precisely when $f(y) \approx_{\mathcal{R}} f(x)$ also holds. Thus, since $\mathcal{R}$ satisfies *sym* means this must be true for *every* $f$ (i.e. every two elements $r$ and $r'$ of $\mathcal{R}$ satisfying $r \approx_{\mathcal{R}} r'$), $\mathcal{R}$ satisfies *sym* precisely when $\approx_{\mathcal{R}}$ is symmetric. Thus we have successfully encoded the concept of symmetry categorically as a morphism.

In general, an object $\mathcal{R}$ is said to satisfy a categorical implication $\mathcal{P} \xrightarrow{\mathit{axiom}} \mathcal{C}$ if all morphisms from $\mathcal{P}$ to $\mathcal{R}$ factor through *axiom*. Informally, $\mathcal{P}$ is the premise of the axiom and $\mathcal{C}$ is the conclusion of the axiom, so an object $\mathcal{R}$ satisfies *axiom* if the conclusion holds whenever the premise holds. Given a set $\mathcal{A}$ of axioms, we can form the full subcategory containing precisely the objects which satisfy every categorical implication *axiom* in $\mathcal{A}$. Figure 4 presents a large variety of common axioms encoded as categorical implications in various $\mathbf{Mono}(\mho)\mathbf{PAlg}(\Omega)\mathbf{Rel}(\Phi)$ categories. The advantage of using these categorical implications is that we can use properties of the morphisms in $\mathcal{A}$ to show that the subcategory formed by $\mathcal{A}$ inherits the factorization structure of the original category. In this way we can guarantee joins of existential types still exists even when we allow existential subtyping to make use of these axioms.

| In $\mathbf{Rel}(\ll: 2)$ | | | |
|---|---|---|---|
| Reflexivity: | $\forall\alpha.$ | $\varnothing \Rightarrow \alpha \ll \alpha$ | $\langle\{\alpha\}, \emptyset\rangle \xrightarrow{\mathit{refl}} \langle\{\alpha\}, \{\alpha \ll \alpha\}\rangle$ |
| Transitivity: | $\forall\alpha, \beta, \gamma.$ | $\alpha \ll \beta \wedge \beta \ll \gamma \Rightarrow \alpha \ll \gamma$ | $\langle\{\alpha, \beta, \gamma\}, \{\alpha \ll \beta, \beta \ll \gamma\}\rangle \xrightarrow{\mathit{trans}} \langle\{\alpha, \beta, \gamma\}, \{\alpha \ll \beta, \beta \ll \gamma, \alpha \ll \gamma\}\rangle$ |
| Anti-Symmetry: | $\forall\alpha, \beta.$ | $\alpha \ll \beta \wedge \beta \ll \alpha \Rightarrow \alpha = \beta$ | $\langle\{\alpha, \beta\}, \{\alpha \ll \beta, \beta \ll \alpha\}\rangle \xrightarrow[\{\alpha, \beta \mapsto \gamma\}]{\mathit{asym}} \langle\{\gamma\}, \{\gamma \ll \gamma\}\rangle$ |
| In $\mathbf{PAlg}(+: 2)$ ($\overline{term}$ means $term$ is defined) | | | |
| Totality: | $\forall x, y.$ | $\varnothing \Rightarrow \overline{x + y}$ | $\langle\{x, y\}, \varnothing\rangle \xrightarrow{\mathit{total}} \langle\{x, y, s\}, \{x + y \mapsto s\}\rangle$ |
| Commutativity: | $\forall x, y.$ | $\overline{x + y} \wedge \overline{y + x} \Rightarrow x + y = y + x$ | $\left\langle \begin{array}{c} \{x, y, s_1, s_2\}, \\ \{x + y \mapsto s_1, y + x \mapsto s_2\} \end{array} \right\rangle \xrightarrow[\{s_1, s_2 \mapsto s\}]{\mathit{comm}} \left\langle \begin{array}{c} \{x, y, s\}, \\ \{x + y, y + x \mapsto s\} \end{array} \right\rangle$ |
| Associativity: | $\forall x, y, z.$ | $\overline{\overline{(x + y) + z} \wedge \overline{x + (y + z)}} \Rightarrow (x + y) + z = x + (y + z)$ | $\left\langle \begin{array}{c} \{x, y, z, s_{x,y}, s_{y,z}, s_1, s_2\}, \\ \left\{ \begin{array}{r} x + y \mapsto s_{x,y}, \\ y + z \mapsto s_{y,z}, \\ s_{x,y} + z \mapsto s_1, \\ x + s_{y,z} \mapsto s_2 \end{array} \right\} \end{array} \right\rangle \xrightarrow[\{s_1, s_2 \mapsto s\}]{\mathit{assoc}} \left\langle \begin{array}{c} \{x, y, z, s_{x,y}, s_{y,z}, s\}, \\ \left\{ \begin{array}{r} x + y \mapsto s_{x,y}, \\ y + z \mapsto s_{y,z}, \\ s_{x,y} + z \mapsto s, \\ x + s_{y,z} \mapsto s \end{array} \right\} \end{array} \right\rangle$ |
| Right Injectivity: | $\forall x, y_1, y_2.$ | $x + y_1 = x + y_2 \Rightarrow y_1 = y_2$ | $\left\langle \begin{array}{c} \{x, y_1, y_2, s\}, \\ \{x + y_1, x + y_2 \mapsto s\} \end{array} \right\rangle \xrightarrow[\{y_1, y_2 \mapsto y\}]{\mathit{rinj}} \left\langle \begin{array}{c} \{x, y, s\}, \\ \{x + y \mapsto s\} \end{array} \right\rangle$ |
| In $\mathbf{PAlg}(0: 0, 1: 0, *: 2)$ ($\overline{term}$ means $term$ is defined) | | | |
| Right Zero: | $\forall x.$ | $\overline{x * 0} \Rightarrow x * 0 = 0$ | $\langle\{x, z, p\}, 0 = z, x * z = p\rangle \xrightarrow[\{p \mapsto z\}]{\mathit{rzero}} \langle\{x, z\}, \{0 \mapsto z, x * z \mapsto z\}\rangle$ |
| Right Identity: | $\forall x.$ | $\overline{x * 1} \Rightarrow x * 1 = x$ | $\langle\{x, o, p\}, 1 = o, x * o = p\rangle \xrightarrow[\{p \mapsto x\}]{\mathit{rident}} \langle\{x, o\}, \{1 \mapsto o, x * o \mapsto x\}\rangle$ |
| In $\mathbf{PAlg}(\texttt{Object}: 0)\mathbf{Rel}(\ll: 2)$ ($\overline{term}$ means $term$ is defined) | | | |
| Bottom: | $\forall\alpha.$ | $\overline{\texttt{Object}} \Rightarrow \alpha \ll \texttt{Object}$ | $\left\langle \begin{array}{c} \{O, \alpha\}, \\ \{\texttt{Object} \mapsto O\}, \\ \varnothing \end{array} \right\rangle \xrightarrow{\mathit{bot}} \left\langle \begin{array}{c} \{O, \alpha\}, \\ \{\texttt{Object} \mapsto O\}, \\ \{\alpha \ll O\} \end{array} \right\rangle$ |
| In $\mathbf{Mono}(\texttt{[]})\mathbf{PAlg}(\texttt{[]}: 1)\mathbf{Rel}(\ll: 2)$ | | | |
| Array Covariance: | $\forall\alpha, \beta.$ | $\alpha \ll \beta \wedge \overline{\alpha\texttt{[]}} \wedge \overline{\beta\texttt{[]}} \Rightarrow \alpha\texttt{[]} \ll \beta\texttt{[]}$ | $\left\langle \begin{array}{c} \{\alpha, \beta, a_\alpha, a_\beta\}, \\ \{\alpha\texttt{[]} \mapsto a_\alpha, \beta\texttt{[]} \mapsto a_\beta\}, \\ \{\alpha \ll \beta\} \end{array} \right\rangle \xrightarrow{\mathit{cov}} \left\langle \begin{array}{c} \{\alpha, \beta, a_\alpha, a_\beta\}, \\ \{\alpha\texttt{[]} \mapsto a_\alpha, \beta\texttt{[]} \mapsto a_\beta\}, \\ \{\alpha \ll \beta, a_\alpha \ll a_\beta\} \end{array} \right\rangle$ |
| Array Covariance′: | $\forall\alpha, \beta.$ | $\alpha\texttt{[]} \ll \beta\texttt{[]} \Rightarrow \alpha \ll \beta$ | $\left\langle \begin{array}{c} \{\alpha, \beta, a_\alpha, a_\beta\}, \\ \{\alpha\texttt{[]} \mapsto a_\alpha, \beta\texttt{[]} \mapsto a_\beta\}, \\ \{a_\alpha \ll a_\beta\} \end{array} \right\rangle \xrightarrow{\mathit{cov'}} \left\langle \begin{array}{c} \{\alpha, \beta, a_\alpha, a_\beta\}, \\ \{\alpha\texttt{[]} \mapsto a_\alpha, \beta\texttt{[]} \mapsto a_\beta\}, \\ \{\alpha \ll \beta, a_\alpha \ll a_\beta\} \end{array} \right\rangle$ |
| Array Inheritance: | $\forall\alpha, \beta.$ | $\alpha \ll \beta\texttt{[]} \Rightarrow \exists\gamma.\ \alpha = \gamma\texttt{[]}$ | $\left\langle \begin{array}{c} \{\alpha, \beta, a_\beta\}, \\ \{\beta\texttt{[]} \mapsto a_\beta\}, \\ \{\alpha \ll a_\beta\} \end{array} \right\rangle \xrightarrow{\mathit{arrinh}} \left\langle \begin{array}{c} \{\alpha, \beta, a_\beta, \gamma\}, \\ \{\beta\texttt{[]} \mapsto a_\beta, \gamma\texttt{[]} \mapsto \alpha\}, \\ \{\alpha \ll a_\beta\} \end{array} \right\rangle$ |

**Fig. 4.** Encodings of common axioms as categorical implications

A full subcategory $\mathbf{S}$ of $\mathbf{C}$ is said to be $\mathcal{E}$-implicational if there is a set of morphisms $\mathcal{A}$ contained within $\mathcal{E}$ such that $\mathbf{S}$ is precisely the full subcategory of objects satisfying all categorical implications in $\mathcal{A}$ [1]. If $\mathbf{C}$ has an $(\mathcal{E}, \mathcal{M})$ factorization structure, then $\mathbf{S}$ is closed under the formation of $\mathcal{M}$ sources [1]: for any $\mathcal{M}$-source $(\mathcal{R} \xrightarrow{m_i} \mathcal{R}_i)_{i \in \mathcal{I}}$, if each $\mathcal{R}_i$ is in $\mathbf{S}$ then so is $\mathcal{R}$ (and consequently each $m_i$ since $\mathbf{S}$ is a *full* subcategory). In particular, for any $(\mathcal{E}, \mathcal{M})$-factorization $\mathcal{R} \xrightarrow{e} (\mathcal{S} \xrightarrow{m_i} \mathcal{T}_i)_{i \in \mathcal{I}}$ of a source *in* $\mathbf{S}$, the new intermediate object $\mathcal{S}$ (and all intermediate morphisms) is also in $\mathbf{S}$. Similarly, the unique $(\mathcal{E}, \mathcal{M})$-diagonalization morphisms $d$ for appropriate squares *in* $\mathbf{S}$ will also be in $\mathbf{S}$ since $\mathbf{S}$ is full. Therefore $\mathbf{S}$ inherits the $(\mathcal{E}, \mathcal{M})$ factorization structure of $\mathbf{C}$ whenever it is formed by categorical implications in $\mathcal{E}$. Also, in this situation if $U : \mathbf{C} \to \mathbf{D}$ has a $(\mathcal{G}, \mathcal{M})$ factorization structure then $U|_{\mathbf{S}} : \mathbf{S} \to \mathbf{D}$ ($U$ restricted to the subcategory $\mathbf{S}$) also has a $(\mathcal{G}, \mathcal{M})$ factorization structure.

### E.3 Designing Existential Bounds

All the categorical implications in Figure 4 are epimorphisms in the relevant $\mathbf{Mono}(\mho)\mathbf{PAlg}(\Omega)\mathbf{Rel}(\Phi)$ category (note that the array inheritance categorical implication *arrinh* is epimorphic because `[]` is required to be injective). We showed earlier that $\mathbf{Mono}(\mho)\mathbf{PAlg}(\Omega)\mathbf{Rel}(\Phi)$ has an (Epimorphism, Initial Mono-Source) factorization structure. Therefore all implicational subcategories using any combination of the categorical implications in Figure 4 inherit this factorization structure.

In our experience, we have been able build our category of bounds for our existential type system by inheriting the (Epimorphism, Initial Mono-Source) factorization structure of some $\mathbf{Mono}(\mho)\mathbf{PAlg}(\Omega)\mathbf{Rel}(\Phi)$. First we construct the objects over some sorting $\Xi$. Then we select some set of categorical implications representing the axioms and inference rules we want to permit existential subtyping to take advantage of. Next we identify the object of constants $\Theta$ and build the category of some class of morphisms under $\Theta$. Lastly, we may specialize the factorization structure on $Cxt$ to restrict constraints of some relations in $\Phi$ to be only on the values of slots in the body of the existential type. Thus we feel the tools we provide are form a flexible and expressive system for designing the bounds of existential types.

At this point we should clarify two possible misconceptions the reader may have. First, the techniques we just provided only construct a category/functor with a useful factorization structure. We do not, however, show that there is an algorithm for actually implementing factorizations in this factorization structure. These tools identify exactly what the mathematical function is, but we are forced to leave the implementation to domain-specific methodologies. The other misconception the reader may have is that the factorization structure provided for the final category of bounds $\mathbf{Bnd}$ formed using the above techniques is not necessarily an (Epimorphism, Initial Mono-Source) factorization structure on $\mathbf{Bnd}$; rather it is the (Epimorphism, Initial Mono-Source) factorization structure inherited from $\mathbf{Mono}(\mho)\mathbf{PAlg}(\Omega)\mathbf{Rel}(\Phi)$. The reason these two factorization structures are distinct is that an epimorphism in $\mathbf{Bnd}$ does not necessarily correspond to an epimorphism in $\mathbf{Mono}(\mho)\mathbf{PAlg}(\Omega)\mathbf{Rel}(\Phi)$. For example, an epimorphism in $\mathbf{MonoAlg}(\Omega)$ is not necessarily an epimorphism in $\mathbf{Alg}(\Omega)$, even though $\mathbf{MonoAlg}(\Omega)$ is an $\mathbf{Alg}(\Omega)$-epimorphism-implicational subcategory of $\mathbf{Alg}(\Omega)$. The point is subtle, but important to understand should the reader attempt to make their own advances beyond the techniques we have provided here.

## F Semantics

Here we reexamine the semantics of existential quantification so that we may prove soundness and monotonicity. First we require some category which we call $\mathbf{Sem}$ for lack of a better name. There must be functors $Sem : \mathbf{Bnd} \to \mathbf{Sem}$ mapping a bound to an the object representing its semantics, and $Abs : \mathbf{Sem} \to \mathbf{Cxt}$ specifying the context for each semantic object, such that $Sem\,;Abs$ equals $Cxt$. Because of this equality, we will continue to unambiguously overload $[-]$ for substitution applied to morphisms in $\mathbf{Sem}$. Often $\mathbf{Sem}$ will be some $\mathbf{Mono}(\mho)\mathbf{PAlg}(\Omega)\mathbf{Rel}(\Gamma)$ category, $\mathbf{Bnd}$ will be some category based on that category, and $\mathbf{Cxt}$ will be $\mathbf{Set}$ or $\mathbf{Rel}(\Gamma)$. Lastly we require an object of constants $\Theta$ in $\mathbf{Sem}$ such that the preordered set $Types(Abs(\Theta))$ forms an abstract domain of simple types over the concrete domain.

We define the abstract domain of existential types with existential subtyping over the abstract domain of simple types as the following:

$$\frac{\mathit{asgn} : Sem(\Delta) \to \Theta \qquad Abs(\Theta) \vdash \tau \leq \tau'[\mathit{asgn}]}{\tau :: \exists \Delta.\tau'}$$

This forms a valid abstract domain: $\tau \leq \tau'$, $\exists \Delta.\tau'' \sqsubseteq \exists \Delta'.\tau'''$, and $\tau' :: \exists \Delta.\tau''$ together imply $\tau :: \exists \Delta'.\tau'''$.

*Proof.* $\exists \Delta.\tau'' \sqsubseteq \exists \Delta'.\tau'''$ holds only if there exists a morphism $\theta : \Delta' \to \Delta$ such that $Cxt(\Delta) \vdash \tau'' \leq \tau'''[Cxt(\theta)]$ holds. $\tau' :: \exists \Delta.\tau''$ holds only if there exists a morphism $\mathit{asgn} : Sem(\Delta) \to \Theta$ such that $Abs(\Theta) \vdash \tau' \leq \tau''[\mathit{asgn}]$ holds. Lastly, $\tau \leq \tau' \leq \tau''[\mathit{asgn}] \leq \tau'''[\theta][\mathit{asgn}] = \tau'''[Sem(\theta)\,;\mathit{asgn}]$ because $[\mathit{asgn}]$ preserves subtypes, $[-]$ is functorial, and $Sem\,;Abs$ equals $Cxt$ (which we have used implicitly). Thus $\tau :: \exists \Delta'.\tau'''$ holds.

We can compose the two layers of abstraction so that existential types form a valid abstract domain over the concrete domain:

$$\frac{\mathit{asgn} : Sem(\Delta) \to \Theta \qquad \sigma :: \tau[\theta]}{\sigma :: \exists \Delta.\tau}$$

Now suppose we have a monotonic dataflow function (for some specific concrete instruction) $flow : Types(Abs(\Theta)) \to Types(Abs(\Theta))$ on the abstract domain of simple types. Monotonicity simply means that this function is actually a morphism $\mathit{flow} : Types(Abs(\Theta)) \to Types(Abs(\Theta))$ in $\mathbf{Prost}$. This dataflow function is *natural* as described informally in the paper if it extends to a *natural transformation* [1] $\mathit{flow} : Types \circ Abs \Rightarrow Types \circ Abs$. This means there must be a dataflow function $\mathit{flow}_{\Sigma} : Types(Abs(\Sigma)) \to Types(Abs(\Sigma))$ for each context $Abs(\Sigma)$ resulting from a semantic object $\Sigma$ in $\mathbf{Sem}$; in other words the flow function is defined independently of the context. Furthermore, the result of the dataflow function should be the same before or after any substitution of semantic objects; that is, for every $\theta : \Sigma \to \Sigma'$ in $\mathbf{Sem}$, $\mathit{flow}_{\Sigma}\,;[\theta]$ must equal $[\theta]\,;\mathit{flow}_{\Sigma'}$. These requirements simply formalize the informal definition of a natural dataflow function provided in the paper and can typically be shown to hold almost trivially.

If a monotonic dataflow function on simple types is natural then we can extend it to the abstract domain of existential types. We define the dataflow function $\mathit{flow}_{\exists}$ on existential types as:

$$\mathit{flow}_{\exists}(\exists \Delta.\tau) = \exists \Delta.\mathit{flow}_{Sem(\Delta)}(\tau)$$

Due to monotonicity and naturality of the original flow function *flow*, this flow function on existential types is also monotonic.

*Proof.* Assume $\exists \Delta.\tau \sqsubseteq \exists \Delta'.\tau'$ holds. Then there exists a morphism $\theta : \Delta' \to \Delta$ such that $Cxt(\Delta) \vdash \tau \leq \tau'[\theta]$ holds. Because *flow* is monotonic, the above subtyping implies that the subtyping $Cxt(\Delta) \vdash \mathit{flow}_{Sem(\Delta)}(\tau) \leq \mathit{flow}_{Sem(\Delta)}(\tau'[\theta])$ holds. Because *flow* is natural it commutes with substitution, so we know that $\mathit{flow}_{Sem(\Delta)}(\tau'[\theta])$ equals $\mathit{flow}_{Sem(\Delta')}(\tau')[\theta]$. By applying this equality to the above subtyping we see that $\theta$ serves as evidence that $\exists \Delta.\mathit{flow}_{Sem(\Delta)}(\tau)$ is an existential subtype of $\exists \Delta'.\mathit{flow}_{Sem(\Delta')}(\tau')$, which are the definitions of $\mathit{flow}_{\exists}(\exists \Delta.\tau)$ and $\mathit{flow}_{\exists}(\exists \Delta'.\tau')$. Therefore $\mathit{flow}_{\exists}$ is monotonic.

If we additionally know that the monotonic natural dataflow function *flow* is a sound abstraction of the concrete function $op$, then we are also guaranteed that $\mathit{flow}_{\exists}$ is also a sound monotonic dataflow function.

*Proof.* Assume we have a concrete state such that $\sigma :: \exists \Delta.\tau$ holds. We need to show that $op(\sigma) :: \mathit{flow}_{\exists}(\exists \Delta.\tau)$ also holds. For $\sigma :: \exists \Delta.\tau$ to hold, there must be an assignment morphism $\mathit{asgn} : Sem(\Delta) \to \Delta$ such that $\sigma :: \tau[\mathit{asgn}]$ holds. Because *flow* is a sound abstraction of $op$, this abstraction informs us that $op(\sigma) :: \mathit{flow}_{\Theta}(\tau[\mathit{asgn}])$ holds. Naturality of *flow* tells us that $\mathit{flow}_{\Theta}(\tau[\mathit{asgn}])$ equals $\mathit{flow}_{Sem(\Delta)}(\tau)[\mathit{asgn}]$. Therefore *asgn* serves as evidence that $op(\sigma) :: \exists \Delta.\mathit{flow}_{Sem(\Delta)}(\tau)$ holds, but this is the definition of $\mathit{flow}_{\exists}(\exists \Delta.\tau)$. Thus, $op(\sigma) :: \mathit{flow}_{\exists}(\exists \Delta.\tau)$ holds, so $\mathit{flow}_{\exists}$ is a sound abstraction of $op$.

Therefore sound monotonic natural dataflow functions over a simple abstract domain easily extend to sound monotonic dataflow functions over an existentially quantified variant of that simple abstract domain, provided permitted mappings between bounds correspond to valid mappings between semantic objects (i.e. the $Sem$ functor must exists). Note that in the above proofs we only used the categorical interpretation of existential types; we did not need the requirements of our framework such as general subtyping or factorization structures. These requirements are only used for inference, not for semantics.