Subtyping for Object Type Constructors

Dominic Duggan[*] Adriana Compagnoni[†]

Abstract

Object type constructors have been introduced as an approach to adding container object types to a language with type inference. Useful subtyping for object type constructors requires a flexible subtype rule for type constructors that is not simply the pointwise extension of subtyping for types. At the same time, subtyping should avoid requiring run-time type checks, as in the Java subtype rule for arrays. An extension of object type constructors is considered to allow this subtyping, extending the kinds of object type constructors with *polarities* that allow this subtyping without jeopardizing soundness.

1 Introduction

Recent years have seen much attention paid to the foundations of typed object oriented programming languages. Less attention has been paid to the types of container objects (for example, vectors, lists and arrays). Such objects provides interfaces that are parameterized by one or more type variables (representing the element type, in the aforesaid examples).

A formal calculus that can be used to model languages with parameterized types is the $\mathbf{F}^\omega$ calculus introduced by Girard [6]. This is a polymorphic λ-calculus where types may contain λ-abstractions, and there is β-reduction at the type level. $\mathbf{F}^\omega$ with subtyping, often denoted $\mathbf{F}^\omega_{<:}$, is a suitable vehicle for studying subtyping in languages with parameterized types. However the usual formulations of $\mathbf{F}^\omega_{<:}$ simply extend subtyping pointwise to type constructors:

$$\sigma_1(\sigma) <: \sigma_2(\sigma) \text{ if } \sigma_1 <: \sigma_2$$

This is the subtype rule used by Abadi and Cardelli in considering a calculus (called $\mathbf{Ob}_{\omega<:\mu}$) that combines $\mathbf{F}^\omega_{<:}$, recursive types and primitive objects [1]. The difficult with this is that, for type-checking with parameterized object types, it is too inflexible. In practice stronger subtypings are necessary. For example we would like subtypings such as:

$$List(T) <: List(T') \text{ if } T <: T'$$

under certain assumptions (for example, lists are immutable). This is not allowed with the constructor subtype rule of $\mathbf{Ob}_{\omega<:\mu}$. On the other hand we must be careful not to allow subtypings such as:

$$Array(T) <: Array(T') \text{ if } T <: T'$$

The Java language requires a run-time type check on assignment to an array element, because it allows this subtype rule.

In earlier work we have introduced *object type constructors*, which are an approach to adding container object types into a language with type inference [5]. The semantics for that language was based on a translation into the calculus $\mathbf{Ob}_{\omega<:\mu}$. The subtyping rule for object type constructors was based on the pointwise rule for type constructors in $\mathbf{F}^\omega_{<:}$. In this paper we extend (a simplified version of) object type constructors with a more powerful subtype rule for type constructors. This rule is based on a system of *polarities* that give more flexible subtypings between constructors, while preserving the soundness of the calculus.

In the next section we review a simplified formulation of object type constructors. In Sect. 3 we extend object type constructors with subtyping based on polarities. In Sect. 5 we provide a subtype-checking algorithm for object type constructors with polarities. In Sect. 6 we give a reduction semantics for the language of object type constructors. In Sect. 7 we consider a semantics for object type constructors in terms of a translation to $\mathbf{F}^\omega$ extended with primitive objects, recursive types and polarities. Sect. 8 considers related work, while Sect. 9 provides our conclusions.

2 Review of Object Type Constructors

In this section we describe the language of object type constructors introduced by Duggan [5]. Our type system is described by this syntax:

$$\begin{array}{rcl}
\sigma & ::= & \alpha \mid (\sigma_1\,\sigma_2) \mid \rightarrow \mid \mathtt{Object}(\delta \mid \overline{\alpha_n} : \overline{\kappa_n})\{D^\sigma\} \mid \\
& & \forall\alpha <: \sigma : \kappa.\,\sigma \mid \forall\alpha <\# \sigma : \kappa.\,\sigma \\
D^\sigma & ::= & \varepsilon \mid D^\sigma_1 D^\sigma_2 \mid \mathtt{var}\ x : \sigma \mid \mathtt{method}\ x : \sigma \\
\kappa & ::= & \mathtt{ty} \mid \kappa \rightarrow \kappa
\end{array}$$

Here type variables α range over both types (such as `int` and `boolean`) and also type *constructors* (such as `list` and `array`). For example the function type arrow $\rightarrow$ is a type constructor of two arguments, the domain and range types; $\sigma_1 \rightarrow \sigma_2$ is infix syntax for the binary application $((\rightarrow\ \sigma_1)\,\sigma_2)$.

$\forall\alpha <: \sigma_1 : \kappa.\,\sigma_2$ denotes the type of a polymorphic function, polymorphic in the type variable α, where α may optionally be constrained to be some subtype of the type bound σ_1. The type parameter α also has a "kind" κ; a kind is intuitively the "type of a type," distinguishing in the type system between types and type constructors. Just as a syntactically valid program must additionally be well-typed to be well-formed, so a syntactically valid type must additionally be well-kinded to be well-formed.

$\mathtt{Object}(\delta \mid \overline{\alpha_n} : \overline{\kappa_n})\{D^\sigma\}$ denotes an object type constructor. Each α_i of kind κ_i denotes a type parameter in a parameterized object type. δ, a type constructor variable of kind

[*]Department of Computer Science, Stevens Institute of Technology, Castle Point on the Hudson, Hoboken, New Jersey 07030. E-mail: {dduggan,abc}@cs.stevens-tech.edu.

$\kappa_1 \rightarrow \cdots \rightarrow \kappa_n \rightarrow \mathtt{ty}$, denotes the fixed point of this potentially recursive object type. D^σ denotes a sequence of instance variable and method type declarations, that may refer to δ and $\alpha_1, \ldots, \alpha_n$. For example a list object type constructor can defined as:

```
type List =
     Object(δ | α:ty){var head:α  var tail:δ(α)}
```

`List` is an abbreviation for an object type *constructor*, with element type α and self type *constructor* δ. For example if we have:

```
val x : List(int)
x.tail.head : int
```

where the latter type arises from unfolding the recursive type of `x` twice, then instantiating the type of the instance variable `head` with the element type of the object (the argument to the object type constructor).

The syntax of the language is given by:

$$\begin{array}{lcl} e & ::= & x \mid (\lambda x:\sigma.\, e) \mid (e_1\, e_2) \mid \\ & & (\Lambda\alpha <: \sigma_1 : \kappa.\, \sigma_2) \mid (\Lambda\alpha <\# \sigma_1 : \kappa.\, \sigma_2) \mid e[\sigma] \mid \\ & & e.x \mid e.x := e' \mid e \rightarrow x \mid e \rightarrow x := e' \mid \\ & & \mathtt{object}(\sigma)\{B\} \\ B & ::= & \varepsilon \mid B_1 B_2 \mid \mathtt{var}\ x = e \mid \mathtt{method}\ x = e \end{array}$$

This language contains both typed functions ($\lambda x:\sigma.\, e$) and polymorphic functions ($\Lambda\alpha <: \sigma : \kappa.\, e$). Application of a polymorphic function to a type is is denoted by $e[\sigma]$. $\mathtt{object}(\sigma)\{B\}$ denotes the creation of an object, with specified type σ, and where B is a sequence of instance variable and method definitions. Operations on objects include accessing instance variables ($e.x$), (functionally) updating instance variables ($e.x := e'$), invoking methods ($e \rightarrow x$), and updating methods ($e \rightarrow x := e'$), where the latter provides a form of "dynamic inheritance." For example a list `cons` operation can be defined as:

```
val cons = Λα:ty. λx:α. λy:List(α).
           object(List){var head=x var tail=y}
val cons : ∀α:ty. α → List(α) → List(α)
```

κ denotes the syntax of kinds for type constructors. Our language includes *higher-order polymorphism*, allowing polymorphic functions to abstract over type constructors. Higher-order polymorphism is necessary when abstracting over object type constructors in bounded type quantification, for example:

```
val cdr =
    Λδ<:List:ty → ty. Λα:ty. λxs:δ(α). xs.tail
val cdr :  ∀δ<:List:ty → ty. ∀α:ty. δ(α) → δ(α)
```

The type system for this language is formalized using a system of type rules for programs, and kind rules for types. In these type and kind rules, Γ is a kind environment binding free type variables. Each type variable has an associated kind, and optionally a subtype bound. Δ is a type environment binding free term variables.

$$\begin{array}{lcl} \Gamma & ::= & \{\} \mid \Gamma_1, \Gamma_2 \mid \alpha : \kappa \mid \alpha <: \sigma : \kappa \mid \alpha <\# \sigma : \kappa \\ \Delta & ::= & \{\} \mid \Delta_1, \Delta_2 \mid x : \sigma \end{array}$$

The type system is specified using judgements of the form:

$\Gamma \vdash \diamond$	kind environment formation
$\Gamma \vdash \Delta$	type environment formation
$\Gamma \vdash \sigma : \kappa$	kind judgement
$\Gamma; \Delta \vdash e : \sigma$	type judgement
$\Gamma \vdash \sigma_1 <: \sigma_2 : \kappa$	subtype judgement
$\Gamma \vdash \sigma_1 <\# \sigma_2 : \kappa$	match judgement

The type rules for the non-object constructs are provided in Fig. 1. The type rules for objects are provided in Fig. 2. Fig. 3 and Fig. 4 in the next section provide the kind and subtype rules for the core of the language (with polarities).

The problem that we wish to address is the provision of a suitable notion of subtyping for object type constructors. In [5], the semantics of object type constructors is based on a translation to an extension of $\mathbf{Ob}_{\omega<:\mu}$ [1], an extension of $\mathbf{F}^\omega_{<:}$ with recursive types and primitive objects. However in this system, subtyping for type constructors is based on the pointwise extension of the subtyping rule for types:

$$\sigma_1(\sigma) <: \sigma_2(\sigma) \quad \text{if} \quad \sigma_1 <: \sigma_2$$

Consider if for example we have `int` $<:$ `real`, then we would reasonably expect to have `List(int)` $<:$ `List(real)`. As long as we only extract elements of the list, it is sound to "forget" that the elements of the list are integers, and "widen" their type to `real`. Of course this is not possible with the pointwise subtype rule for type constructors.

On the other hand, we cannot weaken the subtype rule for type constructors to:

$$\sigma_1(\sigma_1') <: \sigma_2(\sigma_2') \quad \text{if} \quad \sigma_1 <: \sigma_2 \text{ and } \sigma_1' <: \sigma_2'$$

Consider the following example:

```
type List2 =
     Object(δ | α:ty){... method insert:α → δ(α)
                          method max:() → α ...}
val list2 : List2(int) = object(List2){...}
val bogus : List2(real) = list2
bogus.insert(3.142).max()
```

The difficulty is that the `max` method for `list2` is specialized to integers, and uses an ordering operation for integers to compute the maximum element of the list. If we allow `bogus` to bind to `list2`, with type `List2(real)` (widening the element type), then we can insert a non-integer value into the list. Calling `max` will result in integer comparison being applied to a non-integer value. The problem with this example is that α occurs negatively in the type of `insert`; this negative occurrence of the type parameter variable invalidates the widening of the element type for the object type constructor.

There is a related issue when we consider update of instance variables and/or methods. The widening `List(int)` $<:$ `List(real)` is only sound provided instance variables are only accessed, and methods are only accessed. However we wish to consider object types where instance variables and methods can be updated (using the operations $e.x := e'$ and $e \rightarrow x := e'$, respectively). An object type constructor of the form:

$$\mathtt{Object}(\delta \mid \overline{\alpha_n} : \overline{\kappa_n})\{\ldots\ \mathtt{var}\ \mathtt{x} : \sigma \ldots\}$$

where `x` is updatable, corresponds to an object type constructor of the form:

$$\begin{array}{l} \mathtt{Object}\ (\delta \mid \overline{\alpha_n} : \overline{\kappa_n})\ \{\ldots\ \mathtt{var}\ \mathtt{x}_{\mathtt{get}} : \sigma \\ \qquad\qquad\qquad\qquad \mathtt{method}\ \mathtt{x}_{\mathtt{set}} : \sigma \rightarrow \delta(\overline{\alpha_n})\ \ldots\} \end{array}$$

$$\frac{\Delta = \Delta_1, x : \sigma, \Delta_2}{\Gamma; \Delta \vdash x : \sigma} \qquad \text{(VAL VAR)}$$

$$\frac{\Gamma; \Delta, x : \sigma_1 \vdash e : \sigma_2}{\Gamma; \Delta \vdash (\lambda x : \sigma_1.\, e) : (\sigma_1 \to \sigma_2)} \qquad \text{(VAL FUN}\to\text{)}$$

$$\frac{\Gamma; \Delta \vdash e_1 : (\sigma_2 \to \sigma_1) \quad \Gamma; \Delta \vdash e_2 : \sigma_2}{\Gamma; \Delta \vdash (e_1\ e_2) : \sigma_1} \qquad \text{(VAL APPL}\to\text{)}$$

$$\frac{\Gamma, \alpha <: \sigma_1 : \kappa; \Delta \vdash e : \sigma_2}{\Gamma; \Delta \vdash (\Lambda \alpha <: \sigma_1 : \kappa.\, e) : (\forall \alpha <: \sigma_1 : \kappa.\, \sigma_2)} \qquad \text{(VAL FUN<:)}$$

$$\frac{\Gamma; \Delta \vdash e : (\forall \alpha <: \sigma_1 : \kappa.\, \sigma_2) \quad \Gamma \vdash \sigma : \kappa \quad \Gamma \vdash \sigma <: \sigma_1 : \kappa}{\Gamma; \Delta \vdash e[\sigma] : \{\sigma/\alpha\}\sigma_2} \qquad \text{(VAL APPL<:)}$$

$$\frac{\Gamma, \alpha {<\#} \sigma_1 : \kappa; \Delta \vdash e : \sigma_2}{\Gamma; \Delta \vdash (\Lambda \alpha {<\#} \sigma_1 : \kappa.\, e) : (\forall \alpha {<\#} \sigma_1 : \kappa.\, \sigma_2)} \qquad \text{(VAL FUN<\#)}$$

$$\frac{\Gamma; \Delta \vdash e : (\forall \alpha {<\#} \sigma_1 : \kappa.\, \sigma_2) \quad \Gamma \vdash \sigma : \kappa \quad \Gamma \vdash \sigma {<\#} \sigma_1 : \kappa}{\Gamma; \Delta \vdash e[\sigma] : \{\sigma/\alpha\}\sigma_2} \qquad \text{(VAL APPL<\#)}$$

$$\frac{\Gamma; \Delta \vdash e : \sigma \quad \Gamma \vdash \sigma' : \texttt{ty} \quad \Gamma \vdash \sigma <: \sigma' : \texttt{ty}}{\Gamma; \Delta \vdash e : \sigma'} \qquad \text{(VAL SUBSUMPTION)}$$

Figure 1: Core Language Type Rules

$$\frac{\begin{array}{c}\sigma^{\text{obj}} = \texttt{Object}(\delta \mid \overline{\alpha_k} : \overline{\pi_k \kappa_k})\{\texttt{var}\ \overline{x_m \upsilon_m} : \overline{\sigma_m}\ \texttt{method}\ \overline{y_n \upsilon'_n} : \overline{\sigma'_n}\} \\ \kappa = \pi_1 \kappa_1 \to \cdots \to \pi_k \kappa_k \to \texttt{ty} \\ \Gamma \vdash \sigma^{\text{obj}} : \kappa \quad \Gamma, \delta : \kappa, \overline{\alpha_k} : \overline{\kappa_k}; \Delta \vdash e_i : \{\sigma^{\text{obj}}/\delta, \overline{\sigma''_k}/\overline{\alpha_k}\}\sigma_i \text{ for } i = 1, \ldots, m \\ \Gamma, \delta : \kappa, \overline{\alpha_k} : \overline{\kappa_k}; \Delta \vdash e'_j : (\sigma^{\text{obj}}(\overline{\sigma''_k}) \to \{\sigma^{\text{obj}}/\delta, \overline{\sigma''_k}/\alpha_k\}\sigma'_j) \text{ for } j = 1, \ldots, n\end{array}}{\Gamma; \Delta \vdash (\texttt{object}(\sigma^{\text{obj}})\{\texttt{var}\ \overline{x_m} : \overline{\sigma_m} = \overline{e_m}\ \texttt{method}\ \overline{y_n} : \overline{\sigma'_n} = \overline{e'_n}\}) : \sigma^{\text{obj}}(\overline{\sigma''_k})} \qquad \text{(VAL OBJECT)}$$

$$\frac{\begin{array}{c}\sigma^{\text{obj}} = \texttt{Object}(\delta \mid \overline{\alpha_n} : \overline{\pi_n \kappa_n})\{\ldots\ \texttt{var}\ x\upsilon : \sigma'\ \ldots\} \quad \upsilon \in \{+, \circ\} \\ \Gamma; \Delta \vdash e : \sigma(\overline{\sigma_n}) \quad \Gamma \vdash \sigma {<\#} \sigma^{\text{obj}} : \kappa\end{array}}{\Gamma; \Delta \vdash e.x : \{\sigma/\delta, \sigma_1/\alpha_1, \ldots, \sigma_n/\alpha_n\}\sigma'} \qquad \text{(VAL VAR ACCESS)}$$

$$\frac{\begin{array}{c}\sigma^{\text{obj}} = \texttt{Object}(\delta \mid \overline{\alpha_n} : \overline{\pi_n \kappa_n})\{\ldots\ \texttt{var}\ x\upsilon : \sigma'\ \ldots\} \quad \upsilon \in \{-, \circ\} \\ \Gamma; \Delta \vdash e : \sigma(\overline{\sigma_n}) \quad \Gamma \vdash \sigma {<\#} \sigma^{\text{obj}} : \kappa \\ \Gamma; \Delta \vdash e' : \{\sigma/\delta, \sigma_1/\alpha_1, \ldots, \sigma_n/\alpha_n\}\sigma'\end{array}}{\Gamma; \Delta \vdash (e.x := e') : \sigma(\overline{\sigma_n})} \qquad \text{(VAL VAR UPDATE)}$$

$$\frac{\begin{array}{c}\sigma^{\text{obj}} = \texttt{Object}(\delta \mid \overline{\alpha_n} : \overline{\pi_n \kappa_n})\{\ldots\ \texttt{method}\ x\upsilon : \sigma'\ \ldots\} \quad \upsilon \in \{+, \circ\} \\ \Gamma \vdash \sigma^{\text{self}} {<\#} \sigma^{\text{obj}} : \kappa \quad \Gamma; \Delta \vdash e : \sigma^{\text{self}}(\overline{\sigma_n})\end{array}}{\Gamma; \Delta \vdash e \to x : \{\sigma^{\text{self}}/\delta, \overline{\sigma_n}/\overline{\alpha_n}\}\sigma'} \qquad \text{(VAL METHOD INVOKE)}$$

$$\frac{\begin{array}{c}\sigma^{\text{obj}} = \texttt{Object}(\delta \mid \overline{\pi_n \alpha_n} : \overline{\kappa_n})\{\ldots\ \texttt{method}\ x\upsilon : \sigma'\ \ldots\} \quad \upsilon \in \{-, \circ\} \\ \Gamma \vdash \sigma^{\text{self}} {<\#} \sigma^{\text{obj}} : \kappa \quad \Gamma; \Delta \vdash e : \sigma^{\text{self}}(\overline{\sigma_n}) \\ \Gamma; \Delta \vdash e' : (\sigma^{\text{self}}(\overline{\sigma_n}) \to \{\sigma^{\text{self}}/\delta, \overline{\sigma_n}/\overline{\alpha_n}\}\sigma')\end{array}}{\Gamma; \Delta \vdash (e \to x := e') : \sigma^{\text{self}}(\overline{\sigma_n})} \qquad \text{(VAL METHOD UPDATE)}$$

Figure 2: Object Type Rules

This implicit "setter" operation entails negative occurrences of the type parameters in the object interface.

Therefore to have a useful and flexible notion of subtyping for object type constructors, we need to consider two factors:

1. whether type parameters occur negatively or positively (or both, or neither) in the instance variable and method

types; and

2. whether the instance variables or methods can be accessed or updated (or both); simply assuming that all instance variables and methods can be accessed and updated leads to an inflexible type system, where all type parameters occur both negatively and positively, leading to the same inflexible type system summarized in this section.

3 Subtyping of Object Type Constructors

We begin by considering a richer object type system, where we distinguish fields in objects that can be read, or written, or both. As summarized in Sect. 7, the semantics of the language introduced in the previous section is based on a translation into $\mathbf{Ob}_{\omega<:\mu}$ [1]. $\mathbf{Ob}_{\omega<:\mu}$ also allows object types of the form $[x_1 \upsilon_1 : \sigma_1, \ldots, x_n \upsilon_n : \sigma_n]$, where variances υ are given by:

$$\upsilon \quad ::= \quad \circ \quad | \quad + \quad | \quad -$$

with the following implications for operations available on a field, and subtyping on that field:

Variance	Operations	Subtyping
$\circ$	read,write	invariant
$+$	read	covariant
$-$	write	contravariant

We add variances to the syntax of object type constructors in our external language, with useful default variances assumed in their absence. Useful defaults are + for method types and $\circ$ for instance variable types. We extend the syntax of the external language with:

$$D^{\sigma} \quad ::= \quad \varepsilon \quad | \quad D_1^{\sigma} D_2^{\sigma} \quad | \quad \texttt{var } x\upsilon : \sigma \quad | \quad \texttt{method } x\upsilon : \sigma$$

With object field types now decorated with variances, we are now in a position to consider parameterized object types where the type parameters may occur only negatively or only positively (or both). This in turn has implications for the kinds of subtypings we can allow for object type constructors. We would like to allow subtypings between object types that take account of subtypings between arguments to the object type constructors. These subtypings must in turn take account of where the type parameters occur in the object type.

It is perhaps tempting to consider a type rule of the form:

$$\sigma(\sigma_1) <: \sigma(\sigma_2) \quad \text{if} \quad \sigma_1 <: \sigma_2 \text{ and the type parameter of } \sigma \text{ occurs only positively in } \sigma$$

However with higher-order polymorphism this is insufficient; the type constructor in such type expressions may very well be type constructor variables, with no information about where type parameters occur in the types of object types.

The only information we have about a type constructor variable is given by its kind (and optionally we may have a subtype or match upper bound). Therefore we need to extend the notion of kinds with further information: for type constructors, the kind should reflect whether occurrences of the type parameters are only positive or only negative. We extend the kind system with *polarities*:

$$\begin{array}{lcl} \pi & ::= & \circ \quad | \quad + \quad | \quad - \quad | \quad \top \\ \kappa & ::= & \texttt{ty} \quad | \quad \pi\kappa \to \kappa \\ \sigma & ::= & \ldots \quad | \quad \texttt{Object}(\delta \mid \overline{\alpha_k} : \overline{\pi_k \kappa_k})\{D^{\sigma}\} \end{array}$$

with the orderings:

$$\circ <: \pi, \pi <: \top \text{ for any } \pi$$

Intuitively (and assuming that all object fields have positive variance), the list type constructor `List` defined in the previous section has kind $+\texttt{ty} \to \texttt{ty}$, while the `List2` type constructor has kind $\circ\texttt{ty} \to \texttt{ty}$. Then we have:

$$\begin{array}{rll} \texttt{List}(\sigma) <: \texttt{List}(\sigma') & \text{if} & \sigma <: \sigma' \\ \texttt{List2}(\sigma) <: \texttt{List2}(\sigma) & \text{if} & \text{true} \end{array}$$

The polarity $\top$ is provided for the case where a type parameter does not occur at all in an object type constructor.

Fig. 3 provides the kind rules for kinds with polarities. The ordering on polarities gives rise to a contravariant ordering on arrow kinds:

$$(\pi\kappa_1 \to \kappa_2) <: (\pi'\kappa_1' \to \kappa_2') \quad \text{if} \quad \pi' <: \pi, \kappa_1' <: \kappa_1, \kappa_2 <: \kappa_2'$$

For example `List` can be given the less flexible kind $\circ\texttt{ty} \to \texttt{ty}$, since $\circ <: +$ and therefore $(+\texttt{ty} \to \texttt{ty}) <: (\circ\texttt{ty} \to \texttt{ty})$.

Fig. 4 provides the subtype rules for the language without objects. The interesting rule here is CON SUB APPLR, the subtype rule for applications. The subtyping of the argument types in the applications is based on the polarity of the type constructors:

$$\frac{\Gamma \vdash \sigma : (\pi\kappa_2 \to \kappa_1) \qquad \Gamma \vdash \sigma_1 <: \sigma_1' : \pi\kappa_2}{\Gamma \vdash (\sigma\,\sigma_1) <: (\sigma\,\sigma_1') : \kappa_1} \quad \text{(CON SUB APPLR)}$$

The polarized subtype judgement $\Gamma \vdash \sigma <: \sigma' : \pi\kappa$ is in turn axiomatized as follows:

$$\frac{\Gamma \vdash \sigma = \sigma' : \kappa}{\Gamma \vdash \sigma <: \sigma' : \circ\kappa} \quad \text{(POL SUB INVARIANT)}$$

$$\frac{\Gamma \vdash \sigma <: \sigma' : \kappa}{\Gamma \vdash \sigma <: \sigma' : +\kappa} \quad \text{(POL SUB COVARIANT)}$$

$$\frac{\Gamma \vdash \sigma' <: \sigma : \kappa}{\Gamma \vdash \sigma <: \sigma' : -\kappa} \quad \text{(POL SUB CONTRAVARIANT)}$$

$$\frac{\Gamma \vdash \sigma : \kappa \qquad \Gamma \vdash \sigma' : \kappa}{\Gamma \vdash \sigma <: \sigma' : \top\kappa} \quad \text{(POL SUB ANY)}$$

The first of these uses a weak form of equality [8], denoted using the judgement form $\Gamma \vdash \sigma = \sigma' : \kappa$. This weak equality is necessary because of the POL SUB ANY rule, which essentially ignores arguments to a constructor with domain polarity of $\top$. The important rule in the axiomatization of weak equality is that for applications of such constructors:

$$\frac{\Gamma \vdash \sigma : (\top\kappa_1 \to \kappa_2)}{\Gamma \vdash (\sigma\,\sigma_1) = (\sigma\,\sigma_2) : \kappa_2} \quad \text{(CON EQ APPLR } \top\text{)}$$

The remaining rules for equality provide the congruence closure of this equality rule.

To specify formation and subtyping rules for object type constructors with polarities, we need to add several new judgement forms:

$\Gamma \vdash \alpha \in PV(\sigma)$	α occurs positively in σ
$\Gamma \vdash \alpha \in NV(\sigma)$	α occurs negatively in σ
$\Gamma \vdash \alpha : \pi \,\mathbf{in}\, \upsilon\sigma$	α may have polarity π in $\upsilon\sigma$
$\Gamma \vdash \upsilon_1\sigma_1 <: \upsilon_2\sigma_2$	subtyping with variance

The first two of these formalize the notion of positive and negative occurrences in a type. Fig. 5 provides the definition for positive occurrences of a type variable in a type, in the external language with polarities. The rules for negative occurrences are analogous.

The definitions of positive and negative occurrences are used to define allowable polarities for type variables, with the judgement form $\Gamma \vdash \alpha : \pi \,\mathbf{in}\, \upsilon\sigma$. Intuitively a variable with positive polarity can only appear positively in a type, while a variable with negative polarity can only appear negatively. Since these judgements are used in defining the well-formedness of an object type constructor, they must also take account of the variance of the instance variable or method type in which the type variable occurs. Allowable polarities are defined as follows:

$$\Gamma \vdash \alpha : \pi \,\mathbf{in}\, \upsilon\sigma \text{ if } \Gamma \not\vdash \alpha \in PV(\sigma) \text{ and } \Gamma \not\vdash \alpha \in NV(\sigma)$$

$$\Gamma \vdash \alpha : + \,\mathbf{in}\, {+\sigma} \text{ if } \Gamma \not\vdash \alpha \in NV(\sigma)$$

$$\Gamma \vdash \alpha : - \,\mathbf{in}\, {+\sigma} \text{ if } \Gamma \not\vdash \alpha \in PV(\sigma)$$

$$\Gamma \vdash \alpha : - \,\mathbf{in}\, {-\sigma} \text{ if } \Gamma \not\vdash \alpha \in NV(\sigma)$$

$$\Gamma \vdash \alpha : + \,\mathbf{in}\, {-\sigma} \text{ if } \Gamma \not\vdash \alpha \in PV(\sigma)$$

$$\Gamma \vdash \alpha : \circ \,\mathbf{in}\, \upsilon\sigma$$

Fig. 6 provides the type rules for object type formation and subtyping of object types. The object type formation rule (CON OBJECT) ensures that the declared polarities of the type parameters are allowable, using the judgement form $\Gamma \vdash \alpha : \pi \,\mathbf{in}\, \upsilon\sigma$ axiomatized above. The object subtype rule (CON SUB OBJECT) checks for interface containment between object type constructors, using the judgement form $\Gamma \vdash \upsilon_1\sigma_1 <: \upsilon_2\sigma_2$ for subtyping between object field types annotated with variances. Fig. 7 provides the rules for subtyping with variance.

4 A Normal System

A key step in developing a type-checking algorithm is to formulate a "normal" version of the type system, where transitivity in subtyping is folded into the other subtype rules. This normal subtype system is presented in Fig. 8; the remaining elements of the type system (typing, kinding and subkinding, etc) are similar to the original type system. As usual the main differences are the NS SUB VAR and NS SUB PROM rules, that allow a type variable to be promoted to its upper bound. The promotion rule for applications, NS SUB PROM, makes use of the following metafunction:

$$\begin{array}{rcl} \mathit{promote}_\Gamma((\sigma_1\ \sigma_2)) & = & (\mathit{promote}_\Gamma(\sigma_1)\ \sigma_2) \\ \mathit{promote}_\Gamma(\alpha) & = & \sigma \text{ if } \Gamma = \Gamma_1, \alpha <: \sigma : \kappa, \Gamma_2 \\ \mathit{promote}_\Gamma(\sigma) & = & \sigma \text{ otherwise} \end{array}$$

We also combine the NS SUB APPLL and NS SUB APPLR rules, so that the subtype rule for applications allows subtyping simultaneously in both the rator and the rand. Since the polarity for the subtyping of the arguments may be the polarity of the left rator or the right rator, we have two versions of this combined application rule for the normal subtype system.

Lemma 4.1 $\Gamma \vdash \sigma <: \sigma' : \kappa$ *if and only if* $\Gamma \vdash_n \sigma <: \sigma' : \kappa$.

Lemma 4.2 *If* $\Gamma \vdash_n \sigma : \kappa$, *then* $\Gamma \vdash_n \mathit{promote}_\Gamma(\sigma) : \kappa$.

The following lemma is a key technical result for the correctness of type-checking, as well as for transitivity elimination:

Lemma 4.3 *Suppose there is a derivation for* $\Gamma \vdash_n \sigma <: \sigma' : \kappa$. *Then there is a derivation for this judgement, such that no application of* NS SUB APPLL *or* NS SUB APPLR *follows an application of* NS SUB PROM *or* NS SUB VAR *to obtain the left antecedent.*

The verification of this lemma is provided in App. 1.

Lemma 4.4 *Suppose there is a derivation for* $\Gamma \vdash_n \sigma <: \sigma' : \kappa$. *Then there is a derivation for this judgement, such that* NS SUB REFL *is only applied to type variables and object type constructors.*

Theorem 1 (Transitivity Elimination) *If there is a derivation of* $\Gamma \vdash_n \sigma <: \sigma' : \kappa$, *then there is such a derivation that does not make use of the* NS SUB TRANS *rule.*

The key cases in the verification of this theorem are provided in App. 1.

Define a *normal derivation* to be one that does not use NS SUB TRANS, that only applies NS SUB REFL to variables and object type constructors, and such that no application of NS SUB APPLL or NS SUB APPLR follows an application of NS SUB PROM or NS SUB VAR to obtain the left antecedent.

Corollary 4.1 *If there is a derivation of* $\Gamma \vdash_n \sigma <: \sigma' : \kappa$, *then there is a normal derivation for this judgement.*

Lemma 4.5

1. *If* $\Gamma \vdash_n (\forall\alpha <: \sigma_1 : \kappa.\ \sigma_2) <: \sigma' : \mathtt{ty}$, *then* $\sigma' = (\forall\alpha <: \sigma_1' : \kappa.\ \sigma_2')$ *for some* σ_1', σ_2' *such that* $\Gamma \vdash_n \sigma_1 = \sigma_1' : \kappa$ *and* $\Gamma, \alpha <: \sigma_1 : \kappa \vdash_n \sigma_2 <: \sigma_2' : \mathtt{ty}$.

2. *If* $\Gamma \vdash_n (\forall\alpha{<\#}\sigma_1 : \kappa.\ \sigma_2) <: \sigma' : \mathtt{ty}$, *then* $\sigma' = (\forall\alpha{<\#}\sigma_1 : \kappa.\ \sigma_2')$ *for some* σ_1', σ_2' *such that* $\Gamma \vdash_n \sigma_1 = \sigma_1' : \kappa$ *and* $\Gamma, \alpha{<\#}\sigma_1 : \kappa \vdash_n \sigma_2 <: \sigma_2' : \mathtt{ty}$.

3. *If* $\Gamma \vdash_n (\sigma_1 \to \sigma_2) <: \sigma' : \mathtt{ty}$, *then* $\sigma' = (\sigma_1' \to \sigma_2')$ *for some* σ_1', σ_2' *such that* $\Gamma \vdash_n \sigma_1' <: \sigma_1 : \mathtt{ty}$ *and* $\Gamma \vdash_n \sigma_2 <: \sigma_2' : \mathtt{ty}$.

4. *If* $\Gamma \vdash_n (\sigma_0\ \sigma_1\ \ldots\ \sigma_n) <: \sigma' : \kappa$ *then* $\sigma' = (\sigma_0'\ \sigma_1'\ \ldots\ \sigma_n')$ *for some* $\sigma_0', \sigma_1', \ldots, \sigma_n'$, *where* $\sigma_0 = \alpha = \sigma_0'$, *or* σ_0 *and* σ_0' *are object type constructors, and* σ_0 *has kind* $\pi_1\kappa_1 \to \cdots \to \pi_n\kappa_n \to \kappa$, *and* $\Gamma \vdash_n \sigma <: \sigma' : \pi_1\kappa_1 \to \cdots \to \pi_n\kappa_n \to \kappa$, *and* $\Gamma \vdash_n \sigma_i <: \sigma_i' : \pi_i\kappa_i$ *for* $i = 1, \ldots, n$.

5. *If* $\Gamma \vdash_n \mathtt{Object}(\delta \mid \overline{\alpha_k} : \overline{\pi_k\kappa_k})\{\overline{x_{m+n}\upsilon_{m+n}} : \overline{\sigma_{m+n}}\} <: \sigma' : \kappa$, *then* $\sigma' = \mathtt{Object}(\delta' \mid \overline{\alpha_k} : \overline{\pi_k\kappa_k})\{\overline{x_m\upsilon_m'} : \overline{\sigma_m'}\}$ *for some* υ_i' *and* $\sigma_i', i = 1, \ldots, m$, *where* $\kappa = \pi_1\kappa_1 \to \cdots \to \pi_k\kappa_k \to \mathtt{ty}$, *and* $\Gamma, \delta' : \kappa, \delta <: \delta' : \kappa, \overline{\alpha_k} : \overline{\kappa_k} \vdash \upsilon_i\sigma_i <: \upsilon_i'\sigma_i'$ *for* $i = 1, \ldots, m$.

$$\frac{\Gamma = \Gamma_1, \alpha <: \sigma : \kappa, \Gamma_2}{\Gamma \vdash \alpha : \kappa} \qquad \text{(Con Var)}$$

$$\Gamma \vdash \to : ((-\mathtt{ty}) \to (+\mathtt{ty}) \to \mathtt{ty}) \qquad \text{(Con Arrow)}$$

$$\frac{\Gamma \vdash \sigma_1 : (\pi\kappa_2 \to \kappa_1) \quad \Gamma \vdash \sigma_2 : \kappa_2}{\Gamma \vdash (\sigma_1\ \sigma_2) : \kappa_1} \qquad \text{(Con Appl)}$$

$$\frac{\Gamma \vdash \sigma_1 : \kappa \quad \Gamma, \alpha <: \sigma_1 : \kappa \vdash \sigma_2 : \mathtt{ty}}{\Gamma \vdash (\forall \alpha <: \sigma_1 : \kappa.\, \sigma_2) : \mathtt{ty}} \qquad \text{(Con All<:)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \sigma_1 : \kappa \quad \Gamma, \alpha <\# \sigma_1 : \kappa \vdash \sigma_2 : \mathtt{ty} \\ \kappa = \pi_1\kappa_1 \to \cdots \to \pi_n\kappa_n \to \mathtt{ty} \quad \Gamma \vdash \sigma_1 <: \mathtt{Object}(\delta \mid \overline{\alpha_n} : \overline{\pi_n\kappa_n})\{\ \} : \kappa\end{array}}{\Gamma \vdash (\forall \alpha <\# \sigma_1 : \kappa.\, \sigma_2) : \mathtt{ty}} \qquad \text{(Con All<\#)}$$

$$\frac{\Gamma \vdash \sigma : \kappa \quad \vdash \kappa <: \kappa'}{\Gamma \vdash \sigma : \kappa'} \qquad \text{(Con Subsumption)}$$

$$\vdash \kappa <: \kappa \qquad \text{(Kind Sub Refl)}$$

$$\frac{\vdash \kappa_1 <: \kappa_2 \quad \vdash \kappa_2 <: \kappa_3}{\vdash \kappa_1 <: \kappa_3} \qquad \text{(Kind Sub Trans)}$$

$$\frac{\vdash \kappa_1' <: \kappa_1 \quad \vdash \kappa_2 <: \kappa_2' \quad (\pi' <: \pi)}{\vdash (\pi\kappa_1 \to \kappa_2) <: (\pi'\kappa_1' \to \kappa_2')} \qquad \text{(Kind Sub Arrow)}$$

Figure 3: Core Language Kind Rules With Polarities

$$\frac{\Gamma \vdash \sigma = \sigma' : \kappa}{\Gamma \vdash \sigma <: \sigma : \kappa} \qquad \text{(Con Sub Refl)}$$

$$\frac{\Gamma \vdash \sigma_1 <: \sigma_2 : \kappa \quad \Gamma \vdash \sigma_2 <: \sigma_3 : \kappa}{\Gamma \vdash \sigma_1 <: \sigma_3 : \kappa} \qquad \text{(Con Sub Trans)}$$

$$\frac{\Gamma = \Gamma_1, \alpha <: \sigma : \kappa, \Gamma_2}{\Gamma \vdash \alpha <: \sigma : \kappa} \qquad \text{(Con Sub Var)}$$

$$\frac{\Gamma \vdash \sigma_1 <: \sigma_1' : \mathtt{ty} \quad \Gamma \vdash \sigma_2' <: \sigma_2 : \mathtt{ty}}{\Gamma \vdash (\sigma_1 \to \sigma_2) <: (\sigma_1' \to \sigma_2') : \mathtt{ty}} \qquad \text{(Con Sub} \to\text{)}$$

$$\frac{\Gamma \vdash \sigma_1 = \sigma_1' : \kappa \quad \Gamma, \alpha <: \sigma_1 : \kappa \vdash \sigma_2 <: \sigma_2' : \mathtt{ty}}{\Gamma \vdash (\forall \alpha <: \sigma_1 : \kappa.\, \sigma_2) <: (\forall \alpha <: \sigma_1' : \kappa.\, \sigma_2') : \mathtt{ty}} \qquad \text{(Con Sub All<:)}$$

$$\frac{\Gamma \vdash \sigma_1 = \sigma_1' : \kappa \quad \Gamma, \alpha <\# \sigma_1 : \kappa \vdash \sigma_2 <: \sigma_2' : \kappa}{\Gamma \vdash (\forall \alpha <\# \sigma_1 : \kappa.\, \sigma_2) <: (\forall \alpha <\# \sigma_1' : \kappa.\, \sigma_2') : \kappa} \qquad \text{(Con Sub All<\#)}$$

$$\frac{\Gamma \vdash \sigma_1 <: \sigma_1' : (\circ\kappa_2 \to \kappa_1) \quad \Gamma \vdash \sigma : \kappa_2}{\Gamma \vdash (\sigma_1\ \sigma) <: (\sigma_1'\ \sigma) : \kappa_1} \qquad \text{(Con Sub ApplL)}$$

$$\frac{\Gamma \vdash \sigma : (\pi\kappa_2 \to \kappa_1) \quad \Gamma \vdash \sigma_1 <: \sigma_1' : \pi\kappa_2}{\Gamma \vdash (\sigma\ \sigma_1) <: (\sigma\ \sigma_1') : \kappa_1} \qquad \text{(Con Sub ApplR)}$$

Figure 4: Core Language Subtype Rules With Polarities

5 Type-Checking

Our main result is a type-checking algorithm for the type system presented in the previous section. Object type constructors raise complications independent of polarities, involving the interaction between recursive types and impredicative type quantifiers. In general the combination of recursive types and impredicative type quantifiers leads to some type-checking problems,

$$\Gamma \vdash \alpha \in PV(\alpha) \qquad \text{(PV VAR)}$$

$$\frac{\Gamma = \Gamma_1, \beta <: \sigma : \kappa, \Gamma_2 \quad \Gamma \vdash \alpha \in PV(\sigma) \text{ or } \Gamma \vdash \alpha \in NV(\sigma)}{\Gamma \vdash \alpha \in PV(\beta)} \qquad \text{(PV VAR BOUND)}$$

$$\frac{\Gamma \vdash \alpha \in PV(\sigma_1)}{\Gamma \vdash \alpha \in PV((\sigma_1\ \sigma_2))} \qquad \text{(PV APPL)}$$

$$\frac{\Gamma \vdash \sigma_1 : \pi\kappa_1 \to \kappa_2 \quad \pi \in \{+, \circ\} \quad \Gamma \vdash \alpha \in PV(\sigma_2)}{\Gamma \vdash \alpha \in PV((\sigma_1\ \sigma_2))} \qquad \text{(PV APPL POS)}$$

$$\frac{\Gamma \vdash \sigma_1 : \pi\kappa_1 \to \kappa_2 \quad \pi \in \{-, \circ\} \quad \Gamma \vdash \alpha \in NV(\sigma_2)}{\Gamma \vdash \alpha \in PV((\sigma_1\ \sigma_2))} \qquad \text{(PV APPL NEG)}$$

$$\frac{\begin{array}{c}\Gamma, \delta : \pi_1\kappa_1 \to \cdots \to \pi_k\kappa_k \to \mathtt{ty}, \overline{\alpha_k} : \overline{\kappa_k} \vdash \alpha \in PV(\sigma_i) \\ \text{for some } i \in \{1, \ldots, k\}, \upsilon_i \in \{\circ, +\}\end{array}}{\Gamma \vdash \alpha \in PV(\mathtt{Object}(\delta \mid \overline{\alpha_k} : \overline{\pi_k\kappa_k})\{\overline{x_m\upsilon_m} : \overline{\sigma_m}\})} \qquad \text{(PV OBJ POS)}$$

$$\frac{\begin{array}{c}\Gamma, \delta : \pi_1\kappa_1 \to \cdots \to \pi_k\kappa_k \to \mathtt{ty}, \overline{\alpha_k} : \overline{\kappa_k} \vdash \alpha \in NV(\sigma_i) \\ \text{for some } i \in \{1, \ldots, k\}, \upsilon_i \in \{\circ, -\}\end{array}}{\Gamma \vdash \alpha \in PV(\mathtt{Object}(\delta \mid \overline{\alpha_k} : \overline{\pi_k\kappa_k})\{\overline{x_m\upsilon_m} : \overline{\sigma_m}\})} \qquad \text{(PV OBJ NEG)}$$

$$\frac{\Gamma \vdash \alpha \in NV(\sigma) \text{ or } \Gamma \vdash \alpha \in PV(\sigma)}{\Gamma \vdash \alpha \in PV(\forall \beta <: \sigma : \kappa.\, \sigma')} \qquad \text{(PV ALL<: NEG)}$$

$$\frac{\Gamma, \beta <: \sigma : \kappa \vdash \alpha \in PV(\sigma')}{\Gamma \vdash \alpha \in PV(\forall \beta <: \sigma : \kappa.\, \sigma')} \qquad \text{(PV ALL<: POS)}$$

$$\frac{\Gamma \vdash \alpha \in NV(\sigma) \text{ or } \Gamma \vdash \alpha \in PV(\sigma)}{\Gamma \vdash \alpha \in PV(\forall \beta <\# \sigma : \kappa.\, \sigma')} \qquad \text{(PV ALL<\# NEG)}$$

$$\frac{\Gamma, \beta <\# \sigma : \kappa \vdash \alpha \in PV(\sigma')}{\Gamma \vdash \alpha \in PV(\forall \beta <\# \sigma : \kappa.\, \sigma')} \qquad \text{(PV ALL<\# POS)}$$

Figure 5: Positive Occurences of a Variable in a Type

$$\frac{\begin{array}{c}\Gamma, \delta : \kappa, \overline{\alpha_k} : \overline{\kappa_k} \vdash \sigma_i : \mathtt{ty} \text{ for } i = 1, \ldots, m \\ \Gamma, \delta : \kappa, \overline{\alpha_k} : \overline{\kappa_k} \vdash \sigma'_j : \mathtt{ty} \text{ for } j = 1, \ldots, n \qquad \kappa = \pi_1\kappa_1 \to \cdots \to \pi_k\kappa_k \to \mathtt{ty} \\ (\Gamma, \delta : \kappa, \overline{\alpha_k} : \overline{\kappa_k}) \vdash \alpha_h : \pi_h \textbf{ in } \upsilon_i\sigma_i \ \text{ for } i = 1, \ldots, m,\ h = 1, \ldots, k \\ (\Gamma, \delta : \kappa, \overline{\alpha_k} : \overline{\kappa_k}) \vdash \alpha_h : \pi_h \textbf{ in } \upsilon'_j\sigma'_j \ \text{ for } j = 1, \ldots, n,\ h = 1, \ldots, k\end{array}}{\Gamma \vdash (\mathtt{Object}(\delta \mid \overline{\alpha_k} : \overline{\pi_k\kappa_k})\{\mathtt{var}\ \overline{x_m\upsilon_m} : \overline{\sigma_m}\ \mathtt{method}\ \overline{y_n\upsilon'_n} : \overline{\sigma'_n}\}) : \kappa} \qquad \text{(CON OBJECT)}$$

$$\frac{\begin{array}{c}\Gamma, \delta' : \kappa, \delta <: \delta' : \kappa, \overline{\alpha_k} : \overline{\kappa_k} \vdash \upsilon_i\sigma_i <: \upsilon'_i\sigma'_i \text{ for } i = 1, \ldots, m \\ \kappa = \pi_1\kappa_1 \to \cdots \to \pi_k\kappa_k \to \mathtt{ty}\end{array}}{\Gamma \vdash \mathtt{Object}(\delta \mid \overline{\alpha_k} : \overline{\pi_k\kappa_k})\{\overline{x_{m+n}\upsilon_{m+n}} : \overline{\sigma_{m+n}}\} <: \mathtt{Object}(\delta' \mid \overline{\alpha_k} : \overline{\pi_k\kappa_k})\{\overline{x_m\upsilon'_m} : \overline{\sigma'_m}\} : \kappa} \qquad \text{(CON SUB OBJECT)}$$

Figure 6: Object Formation and Subtyping with Polarities

reminiscent of the undecidability of type-checking with a particular form of bounded type quantification [7]. While it is not clear if object type constructors suffer from this fate, we conservatively place the following restriction on types to avoid this undecidability problem:

> A polymorphic type is *simple* if it is of the form $\forall\alpha : \kappa.\ \sigma$ (i.e. with no upper bound on the type parameter). An object type constructor is simple if any polymorphic type nested within it is simple. A general type is simple if any object type constructor occurring in it is simple. A kind environment Γ or type environment Δ is simple if any type bound or type in that environment is simple.

From henceforth we assume that types are simple. Weakening this restriction is a useful direction for further work, although all of the examples we have considered only involved simple types.

Fig. 9 provides the algorithm for checking the subtype relation. The verification of correctness is based on the soundness

$$\frac{\Gamma \vdash \sigma = \sigma' : \mathtt{ty}}{\Gamma \vdash \circ\sigma <: \circ\sigma'} \quad \text{(SUB INVARIANT)}$$

$$\frac{\Gamma \vdash \sigma <: \sigma' : \mathtt{ty} \quad \upsilon \in \{\circ, +\}}{\Gamma \vdash \upsilon\sigma <: +\sigma'} \quad \text{(SUB COVARIANT)}$$

$$\frac{\Gamma \vdash \sigma' <: \sigma : \mathtt{ty} \quad \upsilon \in \{\circ, -\}}{\Gamma \vdash \upsilon\sigma <: -\sigma'} \quad \text{(SUB CONTRAVARIANT)}$$

Figure 7: Subtype Rules With Variance

$$\frac{\Gamma \vdash_n \sigma = \sigma' : \kappa}{\Gamma \vdash_n \sigma <: \sigma : \kappa} \quad \text{(NS SUB REFL)}$$

$$\frac{\Gamma \vdash_n \sigma_1 <: \sigma_2 : \kappa \quad \Gamma \vdash_n \sigma_2 <: \sigma_3 : \kappa}{\Gamma \vdash_n \sigma_1 <: \sigma_3 : \kappa} \quad \text{(NS SUB TRANS)}$$

$$\frac{\Gamma = \Gamma_1, \alpha <: \sigma' : \kappa, \Gamma_2 \quad \Gamma \vdash_n \sigma' <: \sigma : \kappa}{\Gamma \vdash_n \alpha <: \sigma : \kappa} \quad \text{(NS SUB VAR)}$$

$$\frac{\Gamma \vdash_n \sigma_1 <: \sigma_1' : \mathtt{ty} \quad \Gamma \vdash_n \sigma_2' <: \sigma_2 : \mathtt{ty}}{\Gamma \vdash_n (\sigma_1 \to \sigma_2) <: (\sigma_1' \to \sigma_2') : \mathtt{ty}} \quad \text{(NS SUB } \to \text{)}$$

$$\frac{\Gamma \vdash_n \sigma_1 = \sigma_1' : \kappa \quad \Gamma, \alpha <: \sigma_1 : \kappa \vdash_n \sigma_2 <: \sigma_2' : \mathtt{ty}}{\Gamma \vdash_n (\forall \alpha <: \sigma_1 : \kappa.\, \sigma_2) <: (\forall \alpha <: \sigma_1' : \kappa.\, \sigma_2') : \mathtt{ty}} \quad \text{(NS SUB ALL<:)}$$

$$\frac{\Gamma \vdash_n \sigma_1 = \sigma_1' : \kappa \quad \Gamma, \alpha <\# \sigma_1 : \kappa \vdash_n \sigma_2 <: \sigma_2' : \mathtt{ty}}{\Gamma \vdash_n (\forall \alpha <\# \sigma_1 : \kappa.\, \sigma_2) <: (\forall \alpha <\# \sigma_1' : \kappa.\, \sigma_2') : \mathtt{ty}} \quad \text{(NS SUB ALL<\#)}$$

$$\frac{\Gamma \vdash_n \mathit{promote}_\Gamma(\sigma_1\ \sigma_2) <: \sigma : \kappa}{\Gamma \vdash_n (\sigma_1\ \sigma_2) <: \sigma : \kappa} \quad \text{(NS SUB PROM)}$$

$$\frac{\Gamma \vdash_n \sigma_1 <: \sigma_1' : (\circ\kappa_2 \to \kappa_1) \quad \Gamma \vdash_n \sigma_1 : (\pi\kappa_2 \to \kappa_1) \quad \Gamma \vdash_n \sigma_2 <: \sigma_2' : \pi\kappa_2}{\Gamma \vdash_n (\sigma_1\ \sigma_2) <: (\sigma_1'\ \sigma_2') : \kappa_1} \quad \text{(NS SUB APPLL)}$$

$$\frac{\Gamma \vdash_n \sigma_1 <: \sigma_1' : (\circ\kappa_2 \to \kappa_1) \quad \Gamma \vdash_n \sigma_1' : (\pi\kappa_2 \to \kappa_1) \quad \Gamma \vdash_n \sigma_2 <: \sigma_2' : \pi\kappa_2}{\Gamma \vdash_n (\sigma_1\ \sigma_2) <: (\sigma_1'\ \sigma_2') : \kappa_1} \quad \text{(NS SUB APPLR)}$$

$$\frac{\begin{array}{c}\Gamma, \delta' : \kappa, \delta <: \delta' : \kappa, \overline{\alpha_k} : \overline{\kappa_k} \vdash \upsilon_i\sigma_i <: \upsilon_i'\sigma_i' \text{ for } i = 1, \ldots, m \\ \kappa = \pi_1\kappa_1 \to \cdots \to \pi_k\kappa_k \to \mathtt{ty}\end{array}}{\Gamma \vdash_n \mathtt{Object}(\delta \mid \overline{\alpha_k} : \overline{\pi_k\kappa_k})\{\overline{x_{m+n}\upsilon_{m+n}} : \overline{\sigma_{m+n}}\} <: \mathtt{Object}(\delta' \mid \overline{\alpha_k} : \overline{\pi_k\kappa_k})\{\overline{x_m\upsilon_m'} : \overline{\sigma_m'}\} : \kappa} \quad \text{(NS SUB OBJECT)}$$

Figure 8: Normal Subtype Rules With Polarities

and completeness of the normal type system presented in the previous section, and the existence of normal derivations in the normal subtype system.

Theorem 2 *Given* Γ, σ *and* σ' *are simple, then the subtype relation* $\Gamma \vdash \sigma <: \sigma' : \kappa$ *is decidable.*

A type-checking algorithm is defined by a set of type rules that compute the minimal type of a term. Judgements of this algorithm are denoted by $\Gamma;\Delta \vdash_{alg} e : \sigma$. Some parts of this algorithm are provided in Fig. 10. The algorithm makes use of a pair of metafunctions that perform promotion as necessary to promote a type variable to a non-variable upper bound. For accessing and updating fields in an object, the upper bound may be only a match upper bound, justifying the second metafunction. The functions are defined as follows:

$$\begin{array}{rcl}
\mathit{promote}_\Gamma^{<:}((\sigma_1\ \sigma_2)) & = & (\mathit{promote}_\Gamma^{<:}(\sigma_1)\ \sigma_2) \\
\mathit{promote}_\Gamma^{<:}(\alpha) & = & \mathit{promote}_\Gamma^{<:}(\sigma) \\
 & & \text{if } \Gamma = \Gamma_1, \alpha <: \sigma : \kappa, \Gamma_2 \\
\mathit{promote}_\Gamma^{<:}(\sigma) & = & \sigma \text{ otherwise} \\
\mathit{promote}_\Gamma^{<\#}((\sigma_1\ \sigma_2)) & = & (\mathit{promote}_\Gamma^{<\#}(\sigma_1)\ \sigma_2) \\
\mathit{promote}_\Gamma^{<\#}(\alpha) & = & \mathit{promote}_\Gamma^{<\#}(\sigma) \\
 & & \text{if } \Gamma = \Gamma_1, \alpha <\# \sigma : \kappa, \Gamma_2 \\
\mathit{promote}_\Gamma^{<\#}(\alpha) & = & \mathit{promote}_\Gamma^{<\#}(\sigma) \\
 & & \text{if } \Gamma = \Gamma_1, \alpha <: \sigma : \kappa, \Gamma_2 \\
\mathit{promote}_\Gamma^{<\#}(\sigma) & = & \sigma \text{ otherwise}
\end{array}$$

$$\frac{\Gamma \vdash_{alg} \sigma_1' <: \sigma_1 : \mathtt{ty} \quad \Gamma \vdash_{alg} \sigma_2 <: \sigma_2' : \mathtt{ty}}{\Gamma \vdash_{alg} (\sigma_1 \to \sigma_2) <: (\sigma_1' \to \sigma_2') : \mathtt{ty}} \quad (\text{ALGS SUB} \to)$$

$$\frac{\Gamma, \alpha <: \sigma : \kappa \vdash_{alg} \sigma_1 <: \sigma_2 : \mathtt{ty} \quad \Gamma \vdash (\forall \alpha <: \sigma : \kappa.\, \sigma_1) : \mathtt{ty}}{\Gamma \vdash_{alg} (\forall \alpha <: \sigma : \kappa.\, \sigma_1) <: (\forall \alpha <: \sigma : \kappa.\, \sigma_2) : \mathtt{ty}} \quad (\text{ALGS SUB ALL}<:)$$

$$\frac{\Gamma, \alpha <\# \sigma : \kappa \vdash_{alg} \sigma_1 <: \sigma_2 : \mathtt{ty} \quad \Gamma \vdash (\forall \alpha <\# \sigma : \kappa.\, \sigma_1) : \mathtt{ty}}{\Gamma \vdash_{alg} (\forall \alpha <\# \sigma : \kappa.\, \sigma_1) <: (\forall \alpha <\# \sigma : \kappa.\, \sigma_2) : \mathtt{ty}} \quad (\text{ALGS SUB ALL}<\#)$$

$$\frac{\begin{array}{c}\Gamma = \Gamma_1, \alpha <: \sigma : (\pi_1 \kappa_1 \to \cdots \pi_m \kappa_m \to \kappa'), \Gamma_2 \\ \Gamma \vdash_{alg} \sigma_1 <: \sigma_1' : \pi_1 \kappa_1 \quad \ldots \quad \Gamma \vdash_{alg} \sigma_m <: \sigma_m' : \pi_m \kappa_m \quad \kappa' <: \kappa\end{array}}{\Gamma \vdash_{alg} (\alpha\, \sigma_1\, \ldots\, \sigma_m) <: (\alpha\, \sigma_1'\, \ldots\, \sigma_m') : \kappa} \quad (\text{ALGS SUB APPL})$$

$$\frac{\begin{array}{c}\sigma = \mathtt{Object}(\delta \mid \overline{\alpha_k} : \overline{\pi_k \kappa_k})\{\overline{x_{m+n} \upsilon_{m+n}} : \overline{\sigma^1_{m+n}}\} \quad \sigma' = \mathtt{Object}(\delta \mid \overline{\alpha_k} : \overline{\pi_k \kappa_k})\{\overline{x_m \upsilon_m'} : \overline{\sigma^2_m}\} \\ \kappa = \pi_1 \kappa_1 \to \cdots \to \pi_{k_0} \kappa_{k_0} \to \kappa' \quad k_0 \le k \\ \Gamma, \delta' : \kappa, \delta <: \delta' : \kappa, \overline{\alpha_k} : \overline{\kappa_k} \vdash_{alg} \upsilon_i \sigma^1_i <: \upsilon_i' \sigma^2_i : \mathtt{ty} \text{ for } i = 1, \ldots, m \\ \Gamma \vdash_{alg} \sigma_1 <: \sigma_1' : \pi_1 \kappa_1 \quad \ldots \quad \Gamma \vdash_{alg} \sigma_{k_0} <: \sigma_{k_0}' : \pi_{k_0} \kappa_{k_0}\end{array}}{\Gamma \vdash_{alg} (\sigma\, \sigma_1\, \ldots\, \sigma_{k_0}) <: (\sigma'\, \sigma_1'\, \ldots\, \sigma_{k_0}') : \kappa} \quad (\text{ALGS SUB OBJECT})$$

$$\frac{\begin{array}{c}\sigma, \sigma' \text{ are object type constructors of the form } \mathtt{Object}(\delta \mid \overline{\alpha_k} : \overline{\pi_k \kappa_k})\{\ldots\} \\ \kappa = \pi_1 \kappa_1 \to \cdots \to \pi_{k_0} \kappa_{k_0} \to \kappa' \quad k_0 \le k \\ \Gamma \vdash_{alg} \sigma = \sigma' : \kappa \quad \Gamma \vdash_{alg} \sigma_1 <: \sigma_1' : \pi_1 \kappa_1 \quad \ldots \quad \Gamma \vdash_{alg} \sigma_{k_0} <: \sigma_{k_0}' : \pi_{k_0} \kappa_{k_0}\end{array}}{\Gamma \vdash_{alg} (\sigma\, \sigma_1\, \ldots\, \sigma_{k_0}) <: (\sigma'\, \sigma_1'\, \ldots\, \sigma_{k_0}') : \kappa} \quad (\text{ALGS SUB OBJECT REFL})$$

$$\frac{\begin{array}{c}\Gamma = \Gamma_1, \alpha <: \sigma : \kappa', \Gamma_2 \quad \alpha \ne \sigma' \\ \Gamma \vdash_{alg} (\sigma\, \sigma_1\, \ldots\, \sigma_m) <: (\sigma'\, \sigma_1'\, \ldots\, \sigma_n') : \kappa\end{array}}{\Gamma \vdash_{alg} (\alpha\, \sigma_1\, \ldots\, \sigma_m) <: (\sigma'\, \sigma_1'\, \ldots\, \sigma_n') : \kappa} \quad (\text{ALGS SUB PROM})$$

Figure 9: Subtype-Checking Algorithm

Then we have:

Theorem 3 *If $\Gamma; \Delta \vdash e : \sigma$ then $\Gamma; \Delta \vdash_{alg} e : \sigma'$, for some σ' such that $\Gamma \vdash \sigma' <: \sigma : \mathtt{ty}$.*

6 Reduction Semantics

In the next section, we outline a semantics for object type constructors based on a translation to (an extension of) $\mathbf{Ob}_{\omega<:\mu}$. In this section we work instead with a reduction semantics for the language of object type constructors, with reduction rules:

$$(\lambda x : \sigma.\, e_1) e_2 \longrightarrow \{e_2/x\} e_1$$

$$(\Lambda \alpha <: \sigma_1 : \kappa.\, e)[\sigma_2] \longrightarrow \{\sigma_2/\alpha\} e$$

$$(\Lambda \alpha <\# \sigma_1 : \kappa.\, e)[\sigma_2] \longrightarrow \{\sigma_2/\alpha\} e$$

$$(\mathtt{object}(\sigma)\{\ldots\ \mathtt{var}\ \ x = e\ \ldots\}).x \longrightarrow e$$

$$\begin{array}{l}(\mathtt{object}(\sigma)\{\ldots\ \mathtt{var}\ \ x = e\ \ldots\}).x := e' \longrightarrow \\ \qquad\qquad (\mathtt{object}(\sigma)\{\ldots\ \mathtt{var}\ \ x = e'\ \ldots\})\end{array}$$

$$\begin{array}{l}e \to x \longrightarrow e'(e) \\ \quad \text{where } e = \mathtt{object}(\sigma)\{\ldots\ \mathtt{method}\ \ x = e'\ \ldots\}\end{array}$$

$$\begin{array}{l}(\mathtt{object}(\sigma)\{\ldots\ \mathtt{method}\ \ x = e\ \ldots\}) \to x := e' \longrightarrow \\ \qquad\qquad (\mathtt{object}(\sigma)\{\ldots\ \mathtt{method}\ \ x = e'\ \ldots\})\end{array}$$

This description is completed by defining a notion of evaluation contexts, and using this to define a global computation relation.

To verify a subject reduction result for this semantics, we require the following substitution lemma:

Lemma 6.1

1. *Given $\Gamma_1, \alpha : \kappa, \Gamma_2 \vdash \sigma_1 <: \sigma_2 : \kappa$, and $\Gamma_1 \vdash \alpha : \pi$ **in** $+\sigma_i$ for $i = 1$ or $i = 2$, where $\pi = \circ$ if $\alpha \in FV(\Gamma_2)$. Given $\Gamma_1 \vdash \sigma_1' <: \sigma_2' : \pi\kappa$. Then $\Gamma_1, \{\sigma_1'/\alpha\}\Gamma_2 \vdash \{\sigma_1'/\alpha\}\sigma_1 <: \{\sigma_2'/\alpha\}\sigma_2 : \kappa$.*

2. *Given $\Gamma_1, \alpha <: \sigma_1 : \kappa_1, \Gamma_2 \vdash \sigma_2 : \kappa_2$. Given $\Gamma_1 \vdash \sigma : \kappa_1$ and $\Gamma_1 \vdash \sigma <: \sigma_1 : \kappa_1$. Then $\Gamma_1, \{\sigma/\alpha\}\Gamma_2 \vdash \{\sigma/\alpha\}\sigma_2 : \kappa_2$.*

3. *Given $\Gamma; \Delta, x : \sigma_1 \vdash e_1 : \sigma_1 nd$ $\Gamma; \Delta \vdash e_2 : \sigma_2$ Then $\Gamma; \Delta \vdash \{e_2/x\} e_1 : \sigma_1$*

The first part of this lemma relies on the transitivity elimination result for the normal subtype system in the previous section. This is because the polarity condition in the antecedent cannot be applied to the "cut" in uses of transitivity. The verification of the following is based on the verification provided by [1]:

Theorem 4 *If $\{\}; \{\} \vdash e : \sigma$ and $e \longrightarrow e'$, then $\{\}; \{\} \vdash e' : \sigma'$ where $\{\} \vdash \sigma' <: \sigma : \mathtt{ty}$.*

$$\frac{\Delta = \Delta_1, x : \sigma, \Delta_2}{\Gamma; \Delta \vdash_{alg} x : \sigma} \quad \text{(VAL VAR)}$$

$$\frac{\Gamma; \Delta, x : \sigma_1 \vdash_{alg} e : \sigma_2}{\Gamma; \Delta \vdash_{alg} (\lambda x : \sigma_1 .\, e) : (\sigma_1 \to \sigma_2)} \quad \text{(VAL FUN}\to\text{)}$$

$$\frac{\Gamma; \Delta \vdash_{alg} e_1 : \sigma \quad promote_{\Gamma}^{<:}(\sigma) = (\sigma_2 \to \sigma_1) \qquad \Gamma; \Delta \vdash_{alg} e_2 : \sigma_2' \quad \Gamma \vdash_{alg} \sigma_2' <: \sigma_2 : \mathtt{ty}}{\Gamma; \Delta \vdash_{alg} (e_1\ e_2) : \sigma_1} \quad \text{(VAL APPL}\to\text{)}$$

$$\frac{\Gamma, \alpha <: \sigma_1 : \kappa; \Delta \vdash_{alg} e : \sigma_2}{\Gamma; \Delta \vdash_{alg} (\Lambda\alpha <: \sigma_1 : \kappa) : (\forall \alpha <: \sigma_1 : \kappa .\, \sigma_2)} \quad \text{(VAL FUN<:)}$$

$$\frac{\Gamma; \Delta \vdash_{alg} e : \sigma \quad promote_{\Gamma}^{<:}(\sigma) = (\forall \alpha <: \sigma_1 : \kappa .\, \sigma_2) \qquad \Gamma \vdash \sigma' : \kappa \quad \Gamma \vdash_{alg} \sigma' <: \sigma_1 : \kappa}{\Gamma; \Delta \vdash_{alg} e[\sigma'] : \{\sigma'/\alpha\}\sigma_2} \quad \text{(VAL APPL<:)}$$

$$\frac{\sigma^{\text{obj}} = \mathtt{Object}(\delta \mid \overline{\alpha_n} : \overline{\pi_n \kappa_n})\{\ldots\ \mathtt{var}\ x\upsilon : \sigma'\ \ldots\} \quad \upsilon \in \{+, \circ\} \qquad \Gamma; \Delta \vdash_{alg} e : \sigma(\overline{\sigma_n}) \quad promote_{\Gamma}^{<\#}(\sigma) = \sigma^{\text{obj}}}{\Gamma; \Delta \vdash_{alg} e.x : \{\sigma/\delta, \sigma_1/\alpha_1, \ldots, \sigma_n/\alpha_n\}\sigma'} \quad \text{(VAL VAR ACCESS)}$$

Figure 10: Type-Checking Algorithm

7 Translation Semantics

In this section we provide further information about the semantics of object type constructors, by sketching a translation into (an extension of) $\mathbf{Ob}_{\omega<:\mu}$ [1]. This is a simplification of a translation described by [5].

Variances for instance variable and method types in the external language are justified by variances on fields in object types in $\mathbf{Ob}_{\omega<:\mu}$. $\mathbf{Ob}^{\mu\lambda}_{\omega<:\mu}$, introduced in [5], extends $\mathbf{Ob}_{\omega<:\mu}$ with parameterized recursive types, and these form the basis for the translation of recursive object type constructors. Polarities are an extension of $\mathbf{Ob}^{\mu\lambda}_{\omega<:\mu}$; we refer to this extension as $\mathbf{Ob}^{\mu\lambda\pi}_{\omega<:\mu}$. $\mathbf{Ob}_{\omega<:\mu}$ extends subtyping pointwise to type constructors. $\mathbf{Ob}^{\mu\lambda\pi}_{\omega<:\mu}$ incorporates the stronger subtyping for applications based on polarities. $\mathbf{Ob}^{\mu\lambda\pi}_{\omega<:\mu}$ extends the core of the external language with:

$$\begin{array}{rcl} \sigma & ::= & \ldots \mid \lambda\alpha : \pi\kappa .\, \sigma \mid [\overline{x_n \upsilon_n} : \overline{\sigma_n}] \mid \mu\alpha : \kappa .\, \sigma \\ e & ::= & \ldots \mid [\overline{x_n \upsilon_n} = \overline{\varsigma(s : \sigma) e_n}] \mid e.x \mid \\ & & e.x \leftleftharpoons \varsigma(s : \sigma) e' \mid \mathtt{fold}_{\sigma}(e) \mid \mathtt{unfold}(e) \end{array}$$

This adds λ-abstraction at the type level, and primitive objects (with a method update operation) at the term level.

The syntax of type λ-abstractions is extended with polarity on the domain kind ($\lambda\alpha : \pi\kappa .\, \sigma$), with some selected type rules:

$$\frac{\Gamma, \alpha : \kappa_1 \vdash \sigma : \kappa_2 \quad \Gamma, \alpha : \kappa_1 \vdash \alpha : \pi\ \mathbf{in} + \sigma}{\Gamma \vdash (\lambda\alpha : \pi\kappa_1 .\, \sigma) : (\pi\kappa_1 \to \kappa_2)} \quad \text{(CON ABS)}$$

$$\frac{\Gamma, \alpha : \kappa \vdash \sigma : \kappa}{\Gamma \vdash (\mu\alpha : \kappa .\, \sigma) : \kappa} \quad \text{(CON REC)}$$

$$\frac{\Gamma, \alpha : \kappa_1 \vdash \sigma <: \sigma' : \kappa_2}{\Gamma \vdash (\lambda\alpha : \pi\kappa_1 .\, \sigma) <: (\lambda\alpha : \pi\kappa_1 .\, \sigma') : (\pi\kappa_1 \to \kappa_2)} \quad \text{(CON SUB ABS)}$$

$$\frac{\Gamma, \alpha' : \kappa, \alpha <: \alpha' : \kappa \vdash \sigma <: \sigma' : \kappa}{\Gamma \vdash (\mu\alpha : \kappa .\, \sigma) <: (\mu\alpha' : \kappa .\, \sigma') : \kappa} \quad \text{(CON SUB REC)}$$

$$\frac{\Gamma \vdash \sigma = (\mu\alpha : \kappa .\, \sigma') : \kappa \quad \Gamma; \Delta \vdash e : (\{(\mu\alpha : \kappa .\, \sigma')/\alpha\}\sigma')(\overline{\sigma_n})}{\Gamma; \Delta \vdash \mathtt{fold}_{\sigma}(e) : \sigma(\overline{\sigma_n})} \quad \text{(VAL FOLD)}$$

$$\frac{\Gamma; \Delta \vdash e : (\mu\alpha : \kappa .\, \sigma)(\overline{\sigma_n})}{\Gamma; \Delta \vdash \mathtt{unfold}(e) : (\{(\mu\alpha : \kappa .\, \sigma)/\alpha\}\sigma)(\overline{\sigma_n})} \quad \text{(VAL UNFOLD)}$$

The subtype rule for object type constructors CON SUB OBJECT is derived from these subtype rules, based on the translation into $\mathbf{Ob}^{\mu\lambda\pi}_{\omega<:\mu}$. The translation of the object type constructor:

$$\mathtt{Object}(\delta \mid \alpha : \pi\mathtt{ty})\{\mathtt{var}\ \overline{\upsilon_n x_n} : \overline{\sigma_n}\}$$

is the type expression:

$$\mu\delta : (\pi\mathtt{ty} \to \mathtt{ty}).\, \lambda\alpha : \pi\mathtt{ty}.\, [\overline{\upsilon_n x_n} : \overline{[\![\sigma_n]\!]}]$$

where $[\![\sigma_i]\!]$ is the translation of σ_i. (The translation in [5] is somewhat more complicated than this, since it must handle instance variable update where update may change the types of the instance variables.)

To use this internal language as a basis for verifying the soundness of the type system for object type constructors, we can make use of the results of [8] for $\mathbf{F}^{\omega}_{<:}$ with polarities.

8 Related Work

Polarities for subtyping of parameterized object types were first suggested by Cardelli, in an unpublished note [2]. More recently Steffen [8, 9] has investigated $\mathbf{F}^{\omega}_{<:}$ with polarities, and it is useful to compare and contrast our approach and his. There is a great deal of similarity between our arrow kinds $\pi\kappa \to \kappa'$ and his arrow kinds $K \to^{?} K'$. One minor difference is that his polarity $\circ$ denotes our polarity $\top$, while his polarity $\pm$ denotes our polarity $\circ$. Also the direction of inclusion of polarities is reversed, because his inclusion rule for arrow kinds is covariant in the polarities. More substantively, Steffen formulates two version of $\mathbf{F}^{\omega}_{<:}$ with polarities. One of these entails a mutual dependence between the subtyping and kind rules, and so he formulates another version of the type system which separates these through stratification. We work only with a stratified type system. The weak equality rule introduced in Sect. 3 originates with [8].

The major difference between the systems is that Steffen must deal with β-conversion in types in $\mathbf{F}^{\omega}_{<:}$. In contrast we work with the system of object type constructors, without explicit β-conversion. Instead, types are implicitly kept in normal form, with β-reduction to normal form implicit in the elimination type rules. This has enormous implications for the two systems. Without polarities, subtype-checking should terminate on a goal of the form $(\alpha\, \sigma_1 \ldots \sigma_n) <: (\alpha\, \sigma_1 \ldots \sigma_n)$, otherwise the ALGS SUB PROM rule should be used to "promote" the head of the left application. With polarities, this termination check is no longer purely local, instead on a goal $(\alpha\, \sigma_1 \ldots \sigma_n) <: (\alpha\, \sigma'_1 \ldots \sigma'_n)$ the types σ_i, σ'_i must be checked for the appropriate subtype inclusion (This is done by the ALGS SUB APPL rule). If this check fails, promotion must be attempted (Rule ALGS SUB PROM must be applied). This amounts to a backtracking step in the algorithm, leading to an exponential blow-up in the complexity of type-checking. Nor is it correct to always apply ALGS SUB PROM, so this backtracking cannot be avoided.

With our language of object type constructors, on the other hand, we avoid this backtracking by virtue of the fact that our language does not have β-conversion. This means that a goal of the form $(\alpha\, \sigma_1 \ldots \sigma_n) <: (\alpha\, \sigma'_1 \ldots \sigma'_n)$ can never be solved by promoting α on the left to an upper bound that is a λ-abstraction, with the latter then "projecting" a head from its arguments that properly matches the right-hand side. Therefore the algorithm can properly prefer ALGS SUB APPL over ALGS SUB PROM, obviating the need for backtracking.

Object type constructors were introduced as a vehicle for adding container object types to a language with type inference [5]. Although type inference is not a consideration in the current paper, there is one aspect of the design of object type constructors that is crucial to this work. Since type inference with β-conversion is problematic (in general it would appear to require higher-order unification), we do not provide β-conversion in the type language of object type constructors. In practice this means that type abbreviations must be fully applied, in a manner similar to Haskell (which also supports higher-order polymorphism). The absence of β-conversion in types is the crucial facet of object type constructors that avoids the need for back-tracking in the algorithm.

Independently of our work, Pierce has investigated subtyping in $\mathbf{F}^{\omega}_{<:}$ with polarities, and has formulated a version of $\mathbf{F}^{\omega}_{<:}$ with explicit isomorphisms between β-equivalent types that admits a type-checking algorithm without backtracking [Pierce, personal communication].

9 Conclusions

We have described an extension of object type constructors, that in practice appears critical for useful subtyping of container object types. This extension is part of a new language that is being implemented to investigate the usefulness of the mechanisms.
Acknowledgements Thanks to Benjamin Pierce for clarifying the relationship with other work.

References

[1] Martin Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, July 1996.

[2] Luca Cardelli. Notes about $\mathbf{F}^{\omega}_{<:}$. Unpublished manuscript, October 1990.

[3] Adriana Compagnoni. *Higher-Order Subtyping With Intersection Types*. PhD thesis, Cip-Data Koninklijke Bibliotheek, Den Haag, Nijmegen, 1995.

[4] Adriana Compagnoni. Subject reduction and minimal types for higher-order subtyping. Technical Report ECS-LFCS-97-363, Department of Computer Science, University of Edinburgh, 1997.

[5] Dominic Duggan. Object type constructors. Submitted for publication. A preliminary version appeared in the *Workshop on Foundations of Object-Oriented Languages*, New Brunswick, New Jersey, July 1996., 1998.

[6] J.-Y. Girard. *Interprétation Fonctionelle et Elimination des Coupures dans l'Arithmétique d'Ordre Supérieur*. PhD thesis, Thèse de Doctorat d'État, Paris, 1972.

[7] Benjamin Pierce. Bounded quantification is undecidable. In *Proceedings of ACM Symposium on Principles of Programming Languages*. ACM Press, January 1992.

[8] Martin Steffen. *Polarized Higher-Order Subtyping*. PhD thesis, Universität Erlangen-Nürnberg, 1997.

[9] Martin Steffen. Polarized higher-order subtyping (extended abstract). In Zhaohui Luo and Sergei Soloviev, editors, *Electronic Proceedings of the Types working group Workshop on Subtyping, inheritance and modular development of proofs*, August 1997. Available through `http://www.dur.ac.uk/~dcs3apj/cs-types/`.

1 Proof of Transitivity Elimination

Lemma 1.1 *Suppose there is a derivation for $\Gamma \vdash_n \sigma <: \sigma' : \kappa$. Then there is a derivation for this judgement, such that no application of* NS SUB APPLL *or* NS SUB APPLR *follows an application of* NS SUB PROM *or* NS SUB VAR *to obtain the left antecedent.*

PROOF: Given a derivation of the form:

$$\dfrac{\dfrac{\begin{array}{c}\Pi_1\\ \vdots\\ \Gamma \vdash_n \mathit{promote}_\Gamma(\sigma_1) <: \sigma_1' : \pi\kappa_2 \to \kappa_1\end{array}}{\Gamma \vdash_n \sigma_1 <: \sigma_1' : \pi\kappa_2 \to \kappa_1}\text{NS SUB PROM} \qquad \begin{array}{c}\Pi_2\\ \vdots\\ \Gamma \vdash_n \sigma_2 <: \sigma_2' : \pi\kappa_2\end{array}}{\Gamma \vdash_n (\sigma_1\ \sigma_2) <: (\sigma_1'\ \sigma_2') : \kappa_1}\text{NS SUB APPLL}$$

Lemma 4.2 justifies commuting the applications of NS SUB PROM and NS SUB APPLL:

$$\dfrac{\dfrac{\begin{array}{c}\Pi_1\\ \vdots\\ \Gamma \vdash_n \mathit{promote}_\Gamma(\sigma_1) <: \sigma_1' : \pi\kappa_2 \to \kappa_1\end{array} \qquad \begin{array}{c}\Pi_2\\ \vdots\\ \Gamma \vdash_n \sigma_2 <: \sigma_2' : \pi\kappa_2\end{array}}{\Gamma \vdash_n \mathit{promote}_\Gamma(\sigma_1\ \sigma_2) <: (\sigma_1'\ \sigma_2') : \kappa_1}\text{NS SUB PROM}}{\Gamma \vdash_n (\sigma_1\ \sigma_2) <: (\sigma_1'\ \sigma_2') : \kappa_1}\text{NS SUB APPLL}$$

The case for permuting NS SUB VAR and NS SUB APPLL is similar. □

Theorem 5 (Transitivity Elimination) *If there is a derivation of $\Gamma \vdash_n \sigma <: \sigma' : \kappa$, then there is such a derivation that does not make use of the* NS SUB TRANS *rule.*

PROOF: The verification is a standard cut elimination argument, following the approaches of [3, 8]. The non-trivial case is when one or both types in the conclusion of an application of NS SUB TRANS is an application. The complication is when the premises to this application result from the use of NS SUB VAR and NS SUB PROM. We follow the following strategy. First, we assume NS SUB TRANS is applied to two cut-free derivations. Then, for the left premise in the use of transitivity, we use Lemma 4.3 to permute uses of NS SUB PROM and NS SUB VAR with NS SUB APPLL and NS SUB APPLR, and then permute these uses of NS SUB PROM and NS SUB VAR with NS SUB TRANS. This amounts to transforming:

$$\dfrac{\dfrac{\begin{array}{c}\Pi_1\\ \vdots\\ \Gamma \vdash_n \mathit{promote}_\Gamma(\sigma_1\ \sigma_2) <: (\sigma_1'\ \sigma_2') : \kappa\end{array}}{\Gamma \vdash_n (\sigma_1\ \sigma_2) <: (\sigma_1'\ \sigma_2') : \kappa}\text{NS SUB PROM} \qquad \begin{array}{c}\Pi_2\\ \vdots\\ \Gamma \vdash_n (\sigma_1'\ \sigma_2') <: \sigma'' : \kappa\end{array}}{\Gamma \vdash_n (\sigma_1\ \sigma_2) <: \sigma'' : \kappa}\text{NS SUB TRANS}$$

to the following derivation:

$$\dfrac{\dfrac{\begin{array}{c}\Pi_1\\ \vdots\\ \Gamma \vdash_n \mathit{promote}_\Gamma(\sigma_1\ \sigma_2) <: (\sigma_1'\ \sigma_2') : \kappa\end{array} \qquad \begin{array}{c}\Pi_2\\ \vdots\\ \Gamma \vdash_n (\sigma_1'\ \sigma_2') <: \sigma'' : \kappa\end{array}}{\Gamma \vdash_n \mathit{promote}_\Gamma(\sigma_1\ \sigma_2) <: \sigma'' : \kappa}\text{NS SUB TRANS}}{\Gamma \vdash_n (\sigma_1\ \sigma_2) <: \sigma'' : \kappa}\text{NS SUB PROM}$$

The other case to consider is when the right premise to NS SUB TRANS results from an application of NS SUB PROM. By repeatedly applying the previous transformation, we may assume that the left premise of NS SUB TRANS is the result of a sequence of applications of NS SUB APPLL and NS SUB APPLR:

$$\dfrac{\begin{array}{c}\dfrac{\begin{array}{c}\Pi_0\\ \vdots\\ \Gamma \vdash_n \sigma <: \sigma' : \pi_1\kappa_1 \to \cdots \to \pi_n\kappa_n \to \kappa\end{array} \qquad \begin{array}{c}\Pi_1\\ \vdots\\ \Gamma \vdash_n \sigma_1 <: \sigma_1' : \pi_1\kappa_1\end{array}}{\Gamma \vdash_n (\sigma\,\sigma_1) <: (\sigma'\ \sigma_1') : \pi_2\kappa_2 \to \cdots \to \pi_n\kappa_n \to \kappa}\text{NS SUB APPL}\\ \vdots\\ \Gamma \vdash_n (\sigma\,\sigma_1\ \ldots\ \sigma_{n-1}) <: (\sigma'\ \sigma_1'\ \ldots\ \sigma_{n-1}') : (\pi_n\kappa_n \to \kappa)\end{array} \qquad \begin{array}{c}\Pi_n\\ \vdots\\ \Gamma \vdash_n \sigma_n <: \sigma_n' : \pi_n\kappa_n\end{array}}{\Gamma \vdash_n (\sigma\,\sigma_1\ \ldots\ \sigma_n) <: (\sigma'\ \sigma_1'\ \ldots\ \sigma_n') : \kappa}\text{NS SUB APPL}$$

We reason about the possible rules that could have been used to conclude $\Gamma \vdash_n \sigma <: \sigma' : \pi_1\kappa_1 \to \cdots \to \pi_n\kappa_n \to \kappa$.

1. The rule was NS SUB REFL, then $\mathit{promote}_\Gamma(\sigma) = \mathit{promote}_\Gamma(\sigma')$.

2. The rule was NS SUB OBJECT, then $promote_\Gamma(\sigma) = \sigma$ and $promote_\Gamma(\sigma') = \sigma'$.

In either case we obtain a cut-free derivation for $\Gamma \vdash_n promote_\Gamma(\sigma\,\sigma_1\;\ldots\;\sigma_n) <: promote_\Gamma(\sigma'\;\sigma'_1\;\ldots\;\sigma'_n) : \kappa$, of the same height as the original derivation. We use this fact if we have an application of NS SUB TRANS where the right premise is the result of an application of NS SUB PROM:

$$\dfrac{\begin{array}{c}\Pi_1\\ \vdots\\ \Gamma \vdash_n (\sigma_1\;\sigma_2) <: (\sigma'_1\;\sigma'_2) : \kappa\end{array} \qquad \dfrac{\begin{array}{c}\Pi_2\\ \vdots\\ \Gamma \vdash_n promote_\Gamma(\sigma'_1\;\sigma'_2) <: \sigma'' : \kappa\end{array}}{\Gamma \vdash_n (\sigma'_1\;\sigma'_2) <: \sigma'' : \kappa}\text{NS SUB PROM}}{\Gamma \vdash_n (\sigma_1\;\sigma_2) <: \sigma'' : \kappa}\text{NS SUB TRANS}$$

From this we obtain a derivation of the form:

$$\dfrac{\dfrac{\begin{array}{c}\Pi'_1\\ \vdots\\ \Gamma \vdash_n promote_\Gamma(\sigma_1\;\sigma_2) <: promote_\Gamma(\sigma'_1\;\sigma'_2) : \kappa\end{array} \qquad \begin{array}{c}\Pi_2\\ \vdots\\ \Gamma \vdash_n promote_\Gamma(\sigma'_1\;\sigma'_2) <: \sigma'' : \kappa\end{array}}{\Gamma \vdash_n promote_\Gamma(\sigma_1\;\sigma_2) <: \sigma'' : \kappa}\text{NS SUB TRANS}}{\Gamma \vdash_n (\sigma_1\;\sigma_2) <: \sigma'' : \kappa}\text{NS SUB PROM}$$

We are then left with the case where both premises to an application of NS SUB TRANS result from applications of NS SUB APPLL and NS SUB APPLR:

$$\dfrac{\begin{array}{c}\Pi_1\\ \vdots\\ \Gamma \vdash_n (\sigma\,\sigma_1\;\ldots\;\sigma_n) <: (\sigma'\;\sigma'_1\;\ldots\;\sigma'_n) : \kappa\end{array} \qquad \begin{array}{c}\Pi_2\\ \vdots\\ \Gamma \vdash_n (\sigma'\;\sigma'_1\;\ldots\;\sigma'_n) <: (\sigma''\;\sigma''_1\;\ldots;\sigma''_n) : \kappa\end{array}}{\Gamma \vdash_n (\sigma\,\sigma_1\;\ldots\;\sigma_n) <: (\sigma''\;\sigma''_1\;\ldots\;\sigma''_n) : \kappa}\text{NS SUB TRANS}$$

We can assume that each premise is the result of an application of NS SUB REFL or NS SUB OBJECT, following by a series of applications of NS SUB APPLL and NS SUB APPLR. Therefore we have that σ, σ' and σ'' have a common minimum kind $\pi_1\kappa_1 \to \cdots \to \pi_n\kappa_n \to \kappa$, and we have derivations for $\Gamma \vdash_n \sigma <: \sigma' : \pi_1\kappa_1 \to \cdots \to \pi_n\kappa_n \to \kappa$ and $\Gamma \vdash_n \sigma' <: \sigma'' : \pi_1\kappa_1 \to \cdots \to \pi_n\kappa_n \to \kappa$, and $\Gamma \vdash_n \sigma_i <: \sigma'_i : \pi_i\kappa_i$ and $\Gamma \vdash_n \sigma'_i <: \sigma''_i : \pi_i\kappa_i$, for $i = 1,\ldots,n$. Therefore we construct the derivation:

$$\dfrac{\begin{array}{c}\dfrac{\Gamma \vdash_n \sigma <: \sigma'' : \ldots \qquad \dfrac{\Gamma \vdash_n \sigma_1 <: \sigma'_1 : \pi_1\kappa_1 \qquad \Gamma \vdash_n \sigma'_1 <: \sigma''_1 : \pi_1\kappa_1}{\Gamma \vdash_n \sigma_1 <: \sigma''_1 : \pi_1\kappa_1}}{\Gamma \vdash_n (\sigma\,\sigma_1) <: (\sigma''\;\sigma''_1) : \pi_2\kappa_2 \to \cdots \to \pi_n\kappa_n \to \kappa}\\ \vdots\\ \Gamma \vdash_n (\sigma\,\sigma_1\;\ldots\;\sigma_{n-1}) <: (\sigma''\;\sigma''_1\;\ldots\;\sigma''_{n-1}) : (\pi_n\kappa_n \to \kappa)\end{array} \qquad \dfrac{\Gamma \vdash_n \sigma_n <: \sigma'_n : \pi_n\kappa_n \qquad \Gamma \vdash_n \sigma'_n <: \sigma''_n : \pi_n\kappa_n}{\Gamma \vdash_n \sigma_n <: \sigma''_n : \pi_n\kappa_n}}{\Gamma \vdash_n (\sigma\,\sigma_1\;\ldots\;\sigma_n) <: (\sigma''\;\sigma''_1\;\ldots\;\sigma''_n) : \kappa}\text{NS SUB APPLL}$$

□