Type Destructors

Martin Hofmann
Fachbereich Mathematik
Technische Hochschule Darmstadt
Schloßgartenstraße 7
D-64289 Darmstadt, Germany
mh@mathematik.th-darmstadt.de

Benjamin C. Pierce
Computer Science Department
Indiana University
Lindley Hall 215
Bloomington, IN 47405, USA
pierce@cs.indiana.edu

Indiana University
CSCI Technical Report # 502

February 4, 1998

Abstract

We study a variant of System $F_{\leq}$ that integrates and generalizes several existing proposals for calculi with *structural typing rules*. To the usual type constructors ($\rightarrow$, $\times$, All, Some, Rec) we add a number of type *destructors*, each internalizing a useful fact about the subtyping relation. For example, in $F_{\leq}$ with products every closed subtype of a product $\mathtt{S}\times\mathtt{T}$ must itself be a product $\mathtt{S}'\times\mathtt{T}'$ with $\mathtt{S}'$<:S and $\mathtt{T}'$<:T. We internalise this observation by introducing type destructors .1 and .2 and postulating an equivalence $\mathtt{T} =_{\eta} \mathtt{T.1}\times\mathtt{T.2}$ whenever T <: $\mathtt{U}\times\mathtt{V}$ (including, for example, when T is a variable). In other words, every subtype of a product type literally *is* a product type, modulo η-conversion.

Adding type destructors provides a clean solution to the problem of *polymorphic update* without introducing new term formers, new forms of polymorphism, or quantification over type operators. We illustrate this by giving elementary presentations of two well-known encodings of objects, one based on recursive record types and the other based on existential packages.

The formulation of type destructors poses some tricky meta-theoretic problems. We discuss two different variants: an "ideal" system where both constructors and destructors appear in general forms, and a more modest system, $F_{\leq}^{TD}$, which imposes some restrictions in order to achieve a tractable metatheory. The properties of the latter system are developed in detail.

1 Introduction

The search for type-theoretic foundations for object-oriented languages has driven the development of numerous typed lambda-calculi combining polymorphism and subtyping. The prototype of these systems is $F_{\leq}$ [CW85, CG92, CMMS94]. However, in the recent literature, many have observed that $F_{\leq}$ in its pure form is an inadequate framework for object-oriented programming. The problem is that the only way in which subtyping in $F_{\leq}$ can be used in typing terms is via the subsumption rule, which "wastes" some of the information contained in the subtyping relation. In particular, there is no way in $F_{\leq}$ to define *polymorphic update* functions—functions with types like All(X<:T)X$\rightarrow$X that do not behave like the polymorphic identity (or its approximations)—which play an important role in encodings of objects.

To address this shortcoming, several extensions and refinements of $F_{\leq}$ have been proposed, including extensions to higher-order polymorphism [Car90, CL91, PT94, HP95b, PS94, Com94] and a number of special-purpose second-order systems, among them systems with record update [CM91, Car92, FM96, Pol96, HP95a], "structural unfolding" for recursive types [AC96], and "polymorphic repacking" for existential types [Pie96]. What the latter group of extensions have in common is that their soundness is intuitively argued for by using an internalisation of the generation lemma for subtyping: every concrete (closed) subtype of a concrete type T must have the same outermost type former as T.

A simple example in which this principle is applied (popularized by Cardelli [Car95], who attributes it to Abadi) is the following. If X <: $T_1 \times T_2$, then, when X is eventually instantiated with a closed type, this type will be a product whose factors are subtypes of T_1 and T_2, respectively. So it should be sound to assume a function

$$\texttt{mix} \quad \in \quad \texttt{All(X<:Top}\times\texttt{Top) X}\rightarrow\texttt{X}\rightarrow\texttt{X}$$

such that

$$\texttt{mix [}S_1\times S_2\texttt{] e e}' \quad = \quad \texttt{(e.1,e}'\texttt{.2)}.$$

That is, mix takes the first component of its first argument and the second component of its second argument to form a new element of type X. Clearly, this assumes that under the constraint X <: Top×Top the variable X gets instantiated by a product and not by a base type or function type. Although the type system of $F_{\leq}$ ensures that this is indeed the case, $F_{\leq}$ provides no way to make use of this fact to define a function with mix's behavior.

A more interesting example is a refinement of the standard unfolding rule for recursive types. It was used by Abadi, Cardelli, and Viswanathan [ACV96, AC96] to perform method calls in their encoding of objects:

$$\texttt{unfold} \quad \in \quad \texttt{All(X<:Rec(Z)T) X}\rightarrow\texttt{[X/Z]T}.$$

Here the idea is that (assuming monotone subtyping for recursive types) the variable X will eventually be instantiated by some recursive type Rec(Y)S, where Y <: Z ⊢ S <: T. Hence (by subsumption) the ordinary unfolding of x ∈ X also has the type [X/Z]T.

As a final example we mention Pierce's repacking operator for existential types [Pie96]. In order to formulate the existential object encoding [PT94] in a second-order setting, one can introduce a function

```
repack  ∈  All(X<:Some(Z)T)
            (All(Z) All(S<:T) S→S) →
            X → X
```

with the following intended meaning:

```
repack[Some(Z)S] f e  =  open e as [Z,m] in
                          pack f[Z][S]m
                          as Some(Z)S.
```

Intuitively, repack opens its second argument, applies its first argument to it, and repackages the result. The soundness of this operation hinges on the fact that every subtype of an existential type is again an existential type.

In each of these examples one can argue operationally that the addition of the new operators is sound, in the sense that all programs of ground type (possibly containing the new operators) can be reduced to a canonical form [Car95]. In the present paper we present a more radical approach. We propose a new calculus (called $F_{\leq}^{TD}$) in which it is literally the case that every subtype (even a variable) of a product, existential, or recursive type can be regarded as a type with the same shape. As an immediate application, the updating constructs sketched above become definable.

This is done by introducing one or more new type formers, called *type destructors*, for each type constructor that we want to equip with update operations (at present these are cartesian products, existential types, and recursive types). For example, to handle cartesian products we introduce new type formers .1 and .2 (i.e., if T is a type, so are T.1 and T.2), which extract the first and the second component of a cartesian product type. Of course, since not every type is a cartesian product, not every type of the form T.1 is well-formed; for example, Int.1 is not. In order to rule out these unwanted instances, we are lead to stipulate that T.1 is well-formed only if T <: $S_1 \times S_2$ for some types S_1 and S_2.

The type destructors for cartesian products obey a covariant subtyping rule: `S <: T` implies `S.i <: T.i`. In addition, we have β- and η-like type equalities:

$$
\begin{array}{rcll}
(\mathtt{T}_1\times\mathtt{T}_2).i & = & \mathtt{T}_i & \qquad \textsc{Beta-Prod} \\
\mathtt{T} & = & \mathtt{T.1}\times\mathtt{T.2} & \qquad \textsc{Eta-Prod}
\end{array}
$$

Let us see how we can define Abadi's `mix` function using these rules. If $\mathtt{X <: Top}\times\mathtt{Top}$ then `X.1` and `X.2` are well-kinded and we have $\mathtt{X=X.1}\times\mathtt{X.2}$, so

$$\mathtt{fun(X{<:}Top\times Top)\ fun(e{:}X)\ fun(e'{:}X)\ (e.1,e'.2)} \quad \in \quad \mathtt{All(X{<:}Top\times Top)\ X\to X\to X}$$

becomes a valid typing. Notice that without type destructors the above function has the minimal type

$$\mathtt{All(X{<:}Top\times Top)\ X\to X\to (Top\times Top)},$$

which wastes information.

To be able to write interesting (and sound) update functions, we need a type constructor that is invariant in the subtype relation. To this end, we introduce an updatable variant of the cartesian product, written $!\mathtt{T}_1\times\mathtt{T}_2$, which is invariant in its first position and covariant in the second. Here we only need the type destructor `.2`, and the corresponding η-like rule takes the form

$$\mathtt{T} = !\mathtt{S}_1\times\mathtt{T.2} \qquad \textsc{Eta-Prod-Upd}$$

whenever $\mathtt{T <:}\ !\mathtt{S}_1\times\mathtt{S}_2$.

The type destructor together with equation Eta-Prod-Upd allows us to define the following polymorphic update function for updatable products:

$$
\begin{array}{ll}
 & \mathtt{fun(A{<:}Top)\ fun(X{<:}!A\times Top)} \\
 & \ \mathtt{fun(e{:}X)\ fun(a{:}A)} \\
 & \ \mathtt{(a,e.2)} \\
\in & \mathtt{All(A{<:}Top)\ All(X{<:}!A\times Top)\ X\to A\to X}.
\end{array}
$$

Again, without type destructors the minimal type of this function would be

$$\mathtt{All(A{<:}Top)\ All(X{<:}!A\times Top)\ X\to A\to (A\times Top)},$$

which represents a loss of information. This example shows how updatable products provide a way of replacing a component of a compound object by a new value. In Section 4.1 we show how this can be used to encode records with updatable fields and single field update.

1.1 An Ideal System of Type Destructors

Starting from these considerations, we have experimented with a system for type destructors which extends $F_\leq$ with a kinding judgement $\Gamma \vdash \mathtt{T} \in *$ to mean that `T` is a well-formed type in context Γ. Kinding rules for type destructors have subtyping premises; apart from the rules for product types introduced informally above, we have rules for type destructors corresponding to bounded existentials:

$$\frac{\Gamma \vdash \mathtt{S <: Some(X{<:}T_1)T_2}}{\Gamma \vdash \mathtt{EBound(S)} \in *}$$

$$\frac{\Gamma \vdash \mathtt{S <: Some(X{<:}T_1)T_2} \qquad \Gamma \vdash \mathtt{U} \in *}{\Gamma \vdash \mathtt{EBody(U,S)} \in *}$$

In addition, we have β- and η-like equalities for products (as above) and existentials

$$
\begin{array}{rcl}
\mathtt{EBound(Some(X{<:}T_1)T_2)} & = & \mathtt{T_1} \\
\mathtt{EBody(U,Some(X{<:}T_1)T_2)} & = & \mathtt{[U/X]T_2} \\
\mathtt{S} & = & \mathtt{Some(X{<:}EBound(S))\ EBody(X,S)}
\end{array}
$$

(provided the right-hand side is well formed in the final rule). Similar rules can be given for other type formers, including recursive types and even universal and arrow types.

This system looks quite natural and handles all of the above examples. Alas, although the system appears to be sound, its formal metatheory has proved totally unmanageable! The most prominent defect we have found is that β-reduction on well-formed types need not terminate. To see this, suppose that `A` $=$ `Some(X<:Top)Top` and `B` $=$ `EBody(Z,Z)`. The above rules yield `Z<:A` $\vdash$ `B` $\in *$. Now let `C` $=$ `Some(Z<:A)B`. The type expression `EBody(C,C)` is well-formed, but admits an infinite sequence of β-reductions.

This problem makes it very difficult if not impossible to design a complete syntax-directed presentation of subtyping in the style of $F_{\leq}$. One could now accept this lack and look for sound but incomplete semi-algorithms for subtyping and type checking. One would then have to give empirical evidence that these algorithms terminate on many interesting inputs. Indeed, preliminary experiments with the implementation suggest that this might be the case.

1.2 The System $F_{\leq}^{TD}$

In this paper we take, however, a different approach and describe a restricted system, $F_{\leq}^{TD}$, which does have desirable metatheoretic properties such as decidability of all judgements and type soundness. The most prominent difference between $F_{\leq}^{TD}$ and the ideal system is the restriction to *unbounded* existential types. In this way, every type can be reduced to a normal form. Other more technical differences are that we have separated well-formedness from subtyping using a more refined kinding system and that we consider a β-redex like $(\mathtt{T}_1 \times \mathtt{T}_2)\mathtt{.1}$ as definitionally equal to $\mathtt{T}_1$. Finally, we forbid eta-conversion of types in certain positions, notably bounds of universal quantifiers. This simplifies the metatheory and does not seem to restrict the applicability in an essential way. However, we are confident that with some extra work this restriction could be relaxed.

The system $F_{\leq}^{TD}$ contains a destructor for recursive types, and thus allows us to define all of the update operations mentioned above. (We show how to treat the example of structural unfolding for recursive types in Section 3.)

1.3 Translation into $F_{\leq}^{\omega}$

The fragment of $F_{\leq}^{TD}$ without recursive types admits a translation into the higher-order system $F_{\leq}^{\omega}$ [Car90, CL91, PT94, HP95b, PS94, Com94] which is the identity on untyped terms. The details of this translation are a bit heavy notationally, but the basic idea is easy to explain and might improve the reader's intuition for type destructors. Roughly, we translate a variable binding `Z <: T` into a sequence of type variable and type operator variable bindings, followed by a `let`-binding defining `Z` in terms of these newly bound variables. For example, a binding

$$\mathtt{Z} <: \mathtt{Some(X)}\ \mathtt{!X}\times\mathtt{X}\times(\mathtt{X}\to\mathtt{Int})$$

becomes

$$\begin{array}{l} \mathtt{Z}_{\mathtt{EBody(Z).2.1}} <: \mathtt{Fun(X)X} \\ \mathtt{Z}_{\mathtt{EBody(Z).2.2}} <: \mathtt{Fun(X)X}\to\mathtt{Int} \\ \mathtt{Z} = \mathtt{Some(X)}\ \mathtt{!X}\times\mathtt{Z}_{\mathtt{EBody(Z).2.1}}\mathtt{(X)}\times\mathtt{Z}_{\mathtt{EBody(Z).2.2}}\mathtt{(X)}. \end{array}$$

Then, for example, the $F_{\leq}^{TD}$-type `EBody(U,Z).2.1` can be *defined* as $\mathtt{Z}_{\mathtt{EBody(Z).2.1}}\mathtt{(U)}$. Note the similarity between the result of the translation and the original "simple existential encoding" of objects in $F_{\leq}^{\omega}$[PT94, HP95b].

We can thus view (the non-recursive fragment of) $F_{\leq}^{TD}$ as a high-level syntax for such explicit type and operator quantifications. Our experience with a prototype implementation suggests that the use of $F_{\leq}^{TD}$ instead of these explicit quantifications leads to substantially simpler and more readable code.

Extending this translation to recursive types with monotone subtyping would require an extension of $F_{\leq}^{\omega}$ with monotone operator subtyping (cf. [Car90, Ste98]). The ideal system with full bounded existentials does not seem to admit a translation of this kind.

Outline

Section 2 begins the formal treatment of $F_{\leq}^{TD}$ with a definition of its syntax. Section 3 defines the kinding, eta-conversion, subtyping, and typing relations. In Section 4 we reformulate two familiar encodings of objects—the standard recursive-records model and the simple existential model—in $F_{\leq}^{TD}$. Section 5 develops the metatheory of the system in detail. Section 6 sketches a possible denotational semantics for the system. Section 7 offers concluding remarks and some ideas for future work.

2 Syntax

For technical convenience, we split the syntactic class of types into two parts: *neutral types*, consisting of a type variable possibly embedded in a sequence of destructors, and *active types*, which have a concrete type constructor at the head.

			types
`T`	::=	`N`	neutral type
		`A`	active type
			neutral types
`N`	::=	`X`	type variable
		`N.1`	first projection
		`N.2`	second projection
		`EBody(T,N)`	body of an existential type
		`RBody(T,N)`	body of a recursive type
			active types
`A`	::=	`Top`	maximal type
		`Int`	constant type of integers
		$\mathtt{T}_1 \to \mathtt{T}_2$	function type
		$\mathtt{T}_1 \times \mathtt{T}_2$	product type
		$\mathtt{!T}_1 \times \mathtt{T}_2$	updatable product type
		$\mathtt{All(X<:T_1)T}_2$	universal type
		`Some(X)T`	existential type
		`Rec(X)T`	recursive type

The destructors `.1` and `.2` correspond to the product type constructor $\times$ (and, in the case of `.2`, also to the updatable product constructor `!...`$\times$). The destructor `EBody` corresponds to the constructor `Some`; intuitively, `EBody(T,N)` can be read as "The type formed by instantiating the body of the existential type `N` with the value `T` for the bound variable." (For example, consider the type expression `EBody(Int,X)`; if `X` is later instantiated with `Some(Y)Y`$\times$`Y`, then `EBody(Int,X)` will become equivalent to `Int`$\times$`Int`; the definitions of substitution below and the eta-conversion relation in Section 3.2 will make this point clearer.) Similarly, the destructor `RBody` corresponds to the constructor `Rec`.

Notice that we do not provide destructors for all of the constructors. Introducing destructors for contravariant constructors such as $\to$ and `All` would give rise to destructors with contravariant subtyping behavior, which raise difficult metatheoretic problems (requiring backtracking during subtype-checking, etc.). For example, if we were to introduce destructors `Dom` and `Cod` such that $\mathtt{T} =_\eta \mathtt{Dom(T)} \to \mathtt{Cod(T)}$ whenever $\mathtt{T<:S}_1 \to \mathtt{S}_2$ then `Dom` would be contravariant and a goal `X.1 <: Dom(Y)?` could be solved by replacing either `X` or `Y` by its upper bound. On the other hand, we could not think of any useful applications for such contravariant destructors, so we decided to omit them.

Also, notice that—as explained in the introduction—existential types are unbounded in the present system. This is a real restriction: many object encodings can be carried out using only unbounded existentials, but some of the most interesting encodings (e.g. Abadi, Cardelli, and Viswanathan's [ACV96]) do require bounded existential types. Therefore, future research should concentrate on removing this restriction (cf. Section 7).

We will only use the `.1` destructor for non-updatable products: for updatable products it is not needed (if we know that `T <: !U×V`, then we know that the first component of `T` is *exactly* `U`), and allowing it clutters the formal development.

A *typing context* Γ is a list of bindings of the form `x:T`, `X<:T`, or `X:*` such that, whenever $\Gamma = \Gamma'$, `x:T`, Γ'' or $\Gamma = \Gamma'$, `X<:T`, Γ'', all free variables of `T` are bound in Γ'.

			contexts
Γ	::=	$\bullet$	empty context
		Γ, `x:T`	variable binding
		Γ, `X:*`	parameter binding
		Γ, `X<:T`	bounded type variable binding

If `X<:T` occurs in Γ then $\Gamma(\mathtt{X}) \stackrel{\text{def}}{=} \mathtt{T}$; if `X:*` occurs in Γ then $\Gamma(\mathtt{X}) \stackrel{\text{def}}{=} *$. A type variable whose binding has the latter form is called a *parameter*.

2.1 Definition [Substitution]: Since we restrict the application of destructors to neutral types, excluding expressions like $(\mathtt{T}_1\times\mathtt{T}_2)$`.1`, we need to simplify type expressions when we perform a substitution. To do this we define the substitution $[\mathtt{V}/\mathtt{X}](\mathtt{T})$ of type `V` for `X` in `T` by:

$$[\mathtt{V}/\mathtt{X}](\mathtt{N.i}) = \begin{cases} ([\mathtt{V}/\mathtt{X}]\mathtt{N})\mathtt{.i} & \text{if } [\mathtt{V}/\mathtt{X}]\mathtt{N} \text{ neutral} \\ \mathtt{S_i} & \text{if } [\mathtt{V}/\mathtt{X}]\mathtt{N} = \mathtt{S}_1\times\mathtt{S}_2 \text{ or (when } \mathtt{i}{=}2) \ \mathtt{!S}_1\times\mathtt{S}_2 \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$[\mathtt{V}/\mathtt{X}](\mathtt{EBody(T,N)}) = \begin{cases} \mathtt{EBody}([\mathtt{V}/\mathtt{X}]\mathtt{T},[\mathtt{V}/\mathtt{X}]\mathtt{N}) & \text{if } [\mathtt{V}/\mathtt{X}]\mathtt{N} \text{ neutral} \\ [[\mathtt{V}/\mathtt{X}]\mathtt{T}/\mathtt{Y}]\mathtt{S} & \text{if } [\mathtt{V}/\mathtt{X}]\mathtt{N} = \mathtt{Some(Y)S} \\ \text{undefined} & \text{otherwise.} \end{cases}$$

$$[\mathtt{V}/\mathtt{X}](\mathtt{RBody(T,N)}) = \begin{cases} \mathtt{RBody}([\mathtt{V}/\mathtt{X}]\mathtt{T},[\mathtt{V}/\mathtt{X}]\mathtt{N}) & \text{if } [\mathtt{V}/\mathtt{X}]\mathtt{N} \text{ neutral} \\ [[\mathtt{V}/\mathtt{X}]\mathtt{T}/\mathtt{Y}]\mathtt{S} & \text{if } [\mathtt{V}/\mathtt{X}]\mathtt{N} = \mathtt{Rec(Y)S} \\ \text{undefined} & \text{otherwise.} \end{cases}$$

For the other type formers, substitution is defined as usual.

As a notational convenience, the destructors are extended to active types by substitution, e.g.,

$$\begin{aligned} \mathtt{RBody(S,Rec(X)T)} &= [\mathtt{Rec(X)T}/\mathtt{Y}]\mathtt{RBody(S,Y)} \\ &= [\mathtt{S}/\mathtt{X}]\mathtt{T}. \end{aligned}$$

Obviously, these expressions may be undefined.

The term formers of $F_{\leq}^{TD}$ are precisely the familiar ones for the type constructors listed above. Note that there are no extra syntactic forms corresponding to the type destructors.

			terms
`e`	::=	`i`	integer constant
		`x`	variable
		`fun(x:T)e`	function
		$\mathtt{e}_1\ \mathtt{e}_2$	application
		$\mathtt{e}_1\mathtt{,e}_2$	pair
		$\mathtt{!e}_1\mathtt{,e}_2$	updatable pair
		`e.1`	first projection
		`e.2`	second projection
		$\mathtt{fun(X<:T}_1\mathtt{)e}$	polymorphic abstraction
		`e[T]`	polymorphic application
		`fold [R]`	fold a recursive type
		`unfold [R]`	unfold a recursive type
		`pack [S,e] as T`	existential package
		`open e as [X,x] in e`	use of a package

Beta reduction on raw terms is defined as usual as the least reflexive transitive relation $\longrightarrow$ compatible with the term forming operations and closed under the following basic reduction steps:

$$\begin{array}{lcl}
\mathtt{(fun(x{:}T)e)\ e'} & \longrightarrow & \mathtt{[e'/x]e} \\
\mathtt{(fun(X{<:}T)e)[S]} & \longrightarrow & \mathtt{[S/X]e} \\
\mathtt{([!]e_1,e_2).}i & \longrightarrow & \mathtt{e}_i \\
\mathtt{unfold[R_1]\ (fold[R_2]\ e)} & \longrightarrow & \mathtt{e} \\
\mathtt{open\ (pack[S,e]\ as\ T)\ as\ [X,x]\ in\ e'} & \longrightarrow & \mathtt{[S/X][e/x]e'}
\end{array}$$

We sometimes write $\Diamond$ to stand for any of the binary type constructors $\mathtt{!}\ldots\times$, $\times$, and $\rightarrow$.

3 Typing Rules

3.1 Kinding

In order to control the applicability of type destructors we introduce a kinding relation which associates each well-formed type with a type of a special form (called a *kind*) that describes further applicability of destructors. The set of kinds is defined by the following grammar:

$$\begin{array}{lcl}
\mathtt{K} & ::= & \mathtt{Top} \\
 & & \mathtt{X} \\
 & & \mathtt{K_1\times K_2} \\
 & & \mathtt{!T_1\times K_2} \\
 & & \mathtt{Some(X)K} \\
 & & \mathtt{Rec(X)K}
\end{array}$$

To state the kinding relation, we need a variant of the substitution operation that treats variables differently depending on where they occur. Suppose that $\mathtt{K}$ and $\mathtt{L}$ are kinds, $\mathtt{S}$ is a type, and $\mathtt{X}$ is a parameter. Then the substitution of $\mathtt{S}$ and $\mathtt{K}$ for $\mathtt{X}$ in $\mathtt{L}$ is defined as follows (the interesting clause is the one for updatable products):

$$\begin{array}{lcll}
[\mathtt{S,K/X}]\mathtt{Top} & = & \mathtt{Top} & \\
[\mathtt{S,K/X}]\mathtt{X} & = & \mathtt{K} & \\
[\mathtt{S,K/X}]\mathtt{L_1\times L_2} & = & [\mathtt{S,K/X}]\mathtt{L_1}\times[\mathtt{S,K/X}]\mathtt{L_2} & \\
[\mathtt{S,K/X}]\mathtt{!T_1\times L_2} & = & [\mathtt{S/X}]\mathtt{!T_1}\times[\mathtt{S,K/X}]\mathtt{L_2} & \\
[\mathtt{S,K/X}]\mathtt{Some(Y)L_2} & = & \mathtt{Some(Y)}[\mathtt{S,K/X}]\mathtt{L_2} & \text{if } \mathtt{X}\neq\mathtt{Y} \\
[\mathtt{S,K/X}]\mathtt{Rec(Y)L_2} & = & \mathtt{Rec(Y)}[\mathtt{S,K/X}]\mathtt{L_2} & \text{if } \mathtt{X}\neq\mathtt{Y}
\end{array}$$

Intuitively, $[\mathtt{S,K/X}]\mathtt{L}$ is obtained from $\mathtt{L}$ by replacing every occurrence of $\mathtt{X}$ in $\mathtt{L}$ within a left-hand side of an updatable product by $\mathtt{S}$, and every other occurrence by $\mathtt{K}$. The kinding relation $\Gamma\vdash\mathtt{T}\ll\mathtt{K}$ now associates each well-formed type expression $\mathtt{T}$ with an active type $\mathtt{K}$ from which the applicability of destructors can be read off. We say that that $\mathtt{T}$ is well-kinded under Γ and write $\Gamma\vdash\mathtt{T}\in *$ if $\Gamma\vdash\mathtt{T}\ll\mathtt{K}$ for some $\mathtt{K}$.

$$\frac{}{\Gamma\vdash\mathtt{Top}\ll\mathtt{Top}} \qquad \text{(K-Top)}$$

$$\frac{}{\Gamma\vdash\mathtt{Int}\ll\mathtt{Top}} \qquad \text{(K-Base)}$$

$$\frac{\Gamma\vdash\mathtt{T_1}\in * \qquad \Gamma\vdash\mathtt{T_2}\in *}{\Gamma\vdash\mathtt{T_1\rightarrow T_2}\ll\mathtt{Top}} \qquad \text{(K-Arr)}$$

$$\frac{\Gamma\vdash\mathtt{T_1}\ll\mathtt{K_1} \qquad \Gamma\vdash\mathtt{T_2}\ll\mathtt{K_2}}{\Gamma\vdash\mathtt{T_1\times T_2}\ll\mathtt{K_1\times K_2}} \qquad \text{(K-Prod)}$$

$$\frac{\Gamma\vdash\mathtt{T_1}\in * \qquad \Gamma\vdash\mathtt{T_2}\ll\mathtt{K_2}}{\Gamma\vdash\mathtt{!T_1\times T_2}\ll\mathtt{!T_1\times K_2}} \qquad \text{(K-Upd)}$$

$$\frac{\Gamma, \mathtt{X}{:}{*} \vdash \mathtt{T}_2 \ll \mathtt{K}_2}{\Gamma \vdash \mathtt{Some(X)T}_2 \ll \mathtt{Some(X)K}_2} \qquad \text{(K-Some)}$$

$$\frac{\Gamma \vdash \mathtt{T}_1 \in * \qquad \Gamma, \mathtt{X}{<:}\mathtt{T}_1 \vdash \mathtt{T}_2 \in *}{\Gamma \vdash \mathtt{All(X{<:}T}_1\mathtt{)T}_2 \ll \mathtt{Top}} \qquad \text{(K-All)}$$

$$\frac{\Gamma, \mathtt{X}{:}{*} \vdash \mathtt{T} \ll \mathtt{K}}{\Gamma \vdash \mathtt{Rec(X)T} \ll \mathtt{Rec(X)K}} \qquad \text{(K-Rec)}$$

$$\frac{\Gamma(\mathtt{X}) = *}{\Gamma \vdash \mathtt{X} \ll \mathtt{X}} \qquad \text{(K-Param)}$$

$$\frac{\Gamma \vdash \Gamma(\mathtt{X}) \ll \mathtt{K}}{\Gamma \vdash \mathtt{X} \ll \mathtt{K}} \qquad \text{(K-Var)}$$

$$\frac{\Gamma \vdash \mathtt{N} \ll \mathtt{K}_1 \times \mathtt{K}_2}{\Gamma \vdash \mathtt{N.1} \ll \mathtt{K}_1} \qquad \text{(K-Fst)}$$

$$\frac{\Gamma \vdash \mathtt{N} \ll \mathtt{K}_1 \times \mathtt{K}_2}{\Gamma \vdash \mathtt{N.2} \ll \mathtt{K}_2} \qquad \text{(K-Snd)}$$

$$\frac{\Gamma \vdash \mathtt{N} \ll \mathtt{!T}_1 \times \mathtt{K}_2}{\Gamma \vdash \mathtt{N.2} \ll \mathtt{K}_2} \qquad \text{(K-Snd-Upd)}$$

$$\frac{\Gamma \vdash \mathtt{T} \ll \mathtt{K}_1 \qquad \Gamma \vdash \mathtt{N} \ll \mathtt{Some(X)K}_2}{\Gamma \vdash \mathtt{EBody(T,N)} \ll [\mathtt{T},\mathtt{K}_1/\mathtt{X}]\mathtt{K}_2} \qquad \text{(K-EBody)}$$

$$\frac{\Gamma \vdash \mathtt{T} \ll \mathtt{K}_1 \qquad \Gamma \vdash \mathtt{N} \ll \mathtt{Rec(X)K}_2}{\Gamma \vdash \mathtt{RBody(T,N)} \ll [\mathtt{T},\mathtt{K}_1/\mathtt{X}]\mathtt{K}_2} \qquad \text{(K-RBody)}$$

The most interesting rules are K-Upd, K-EBody, and K-RBody. K-EBody, for example, can be read as follows: "If `N` is bounded by an existential type of the form $\mathtt{Some(X)K}_2$ and `T` is well-kinded, then the destructor application `EBody(T,N)` is well-kinded and has the form $\mathtt{K}_2$, with $\mathtt{K}_1$ (or `T` in left-hand sides of updatable products) substituted for the bound variable `X`."

Note that kinding is a (partial) function: If $\Gamma \vdash \mathtt{S} \ll \mathtt{K}$ and $\Gamma \vdash \mathtt{S} \ll \mathtt{L}$, then $\mathtt{K} = \mathtt{L}$.

3.2 Eta-Conversion

The *eta-conversion* relation between types, written $\Gamma \vdash \mathtt{S} =_\eta \mathtt{T}$, is the least equivalence relation closed under the following rules,

$$\frac{\Gamma \vdash \mathtt{N} \ll \mathtt{K}_1 \times \mathtt{K}_2}{\Gamma \vdash \mathtt{N} =_\eta \mathtt{N.1} \times \mathtt{N.2}} \qquad \text{(Eta-Prod)}$$

$$\frac{\Gamma \vdash \mathtt{N} \ll \mathtt{!T} \times \mathtt{K}}{\Gamma \vdash \mathtt{N} =_\eta \mathtt{!T} \times \mathtt{N.2}} \qquad \text{(Eta-Upd)}$$

$$\frac{\Gamma \vdash \mathtt{N} \ll \mathtt{Some(X)K}}{\Gamma \vdash \mathtt{N} =_\eta \mathtt{Some(X)EBody(X,N)}} \qquad \text{(Eta-Some)}$$

$$\frac{\Gamma \vdash \mathtt{N} \ll \mathtt{Rec(X)K}}{\Gamma \vdash \mathtt{N} =_\eta \mathtt{Rec(X)RBody(X,N)}} \quad \text{(Eta-Rec)}$$

plus congruence rules for all the type formers except the bounds of universal quantifiers and the first (substitutive) arguments of EBody and RBody. (The prohibition of eta-conversion in these positions is a somewhat ad-hoc restriction, needed in our proof of completeness of the syntax-directed presentation of eta-conversion and subtyping. We conjecture that it can be relaxed.)

3.2.1 Lemma [Eta-congruence preserves kinding]: If $\Gamma \vdash \mathtt{S} =_\eta \mathtt{T}$ and $\Gamma \vdash \mathtt{S} \ll \mathtt{K}$, then $\Gamma \vdash \mathtt{T} \ll \mathtt{L}$ and $\Gamma \vdash \mathtt{K} =_\eta \mathtt{L}$.

(One might expect that, with the restricted definition of eta-conversion that we're using at the moment, this property could be made even stronger: $\mathtt{K} = \mathtt{L}$. But this is still not the case, for example, when $\mathtt{S} = \mathtt{!(X.1\times X.2)\times Top}$ and $\mathtt{T} = \mathtt{!X\times Top}$.)

Proof: By induction on a derivation of $\Gamma \vdash \mathtt{S} =_\eta \mathtt{T}$, with a case analysis on the final rule used. For example, suppose the final rule is Eta-Some—i.e., we have

$$\begin{array}{lcl} \mathtt{S} & = & \mathtt{N} \\ \mathtt{T} & = & \mathtt{Some(X)EBody(X,N)} \\ \mathtt{K} & = & \mathtt{Some(X)K'}. \end{array}$$

By the kinding rules (using the assumption $\Gamma \vdash \mathtt{N} \ll \mathtt{K}$), we have:

$$\cfrac{\cfrac{\Gamma, \mathtt{X}{:}{*} \vdash \mathtt{X} \ll \mathtt{X} \qquad \Gamma, \mathtt{X}{\in}{*} \vdash \mathtt{N} \ll \mathtt{Some(X)K'}}{\Gamma, \mathtt{X}{:}{*} \vdash \mathtt{EBody(X,N)} \ll [\mathtt{X},\mathtt{X}/\mathtt{X}]\mathtt{K'}}}{\Gamma \vdash \mathtt{T} \ll \mathtt{Some(X)}[\mathtt{X},\mathtt{X}/\mathtt{X}]\mathtt{K'}}$$

Finally, note that $[\mathtt{X},\mathtt{X}/\mathtt{X}]\mathtt{K'}$ is always defined and equals $\mathtt{K'}$. ∎

3.3 Subtyping

The subtyping rules for the active type formers are the same as in (the Kernel Fun variant of) $F_\leq$; that is to say, ordinary products are covariant in both arguments, function spaces are contravariant in the first position and covariant in the second, universal quantifiers and updatable products are invariant in the first position and covariant in the second, unbounded existentials are covariant, and recursive types obey the monotone subtyping rule mentioned in the introduction.

$$\frac{\Gamma \vdash \mathtt{S} \in *}{\Gamma \vdash \mathtt{S} <: \mathtt{S}} \quad \text{(S-Refl)}$$

$$\frac{\Gamma \vdash \mathtt{S} <: \mathtt{U} \qquad \Gamma \vdash \mathtt{U} <: \mathtt{T}}{\Gamma \vdash \mathtt{S} <: \mathtt{T}} \quad \text{(S-Trans)}$$

$$\frac{\Gamma \vdash \mathtt{S} \in *}{\Gamma \vdash \mathtt{S} <: \mathtt{Top}} \quad \text{(S-Top)}$$

$$\frac{\Gamma \vdash \Gamma(\mathtt{X}) \in *}{\Gamma \vdash \mathtt{X} <: \Gamma(\mathtt{X})} \quad \text{(S-Var)}$$

$$\frac{\Gamma \vdash \mathtt{T}_1 <: \mathtt{S}_1 \qquad \Gamma \vdash \mathtt{S}_2 <: \mathtt{T}_2}{\Gamma \vdash \mathtt{S}_1 \to \mathtt{S}_2 <: \mathtt{T}_1 \to \mathtt{T}_2} \quad \text{(S-Arrow)}$$

$$\frac{\Gamma,\ \mathtt{X}{<:}\mathtt{U}_1 \vdash \mathtt{S}_2 \mathrel{<:} \mathtt{T}_2}{\Gamma \vdash \mathtt{All(X<:U1)S}_2 \mathrel{<:} \mathtt{All(X<:U1)T}_2} \qquad \text{(S-All)}$$

$$\frac{\Gamma,\ \mathtt{X}{:}{*} \vdash \mathtt{S} \mathrel{<:} \mathtt{T}}{\Gamma \vdash \mathtt{Some(X)S} \mathrel{<:} \mathtt{Some(X)T}} \qquad \text{(S-Some)}$$

$$\frac{\Gamma,\ \mathtt{Y}{:}{*},\ \mathtt{X}{<:}\mathtt{Y} \vdash \mathtt{S} \mathrel{<:} \mathtt{T}}{\Gamma \vdash \mathtt{Rec(X)S} \mathrel{<:} \mathtt{Rec(Y)T}} \qquad \text{(S-Rec)}$$

$$\frac{\Gamma \vdash \mathtt{S}_1 \mathrel{<:} \mathtt{T}_1 \qquad \Gamma \vdash \mathtt{S}_2 \mathrel{<:} \mathtt{T}_2}{\Gamma \vdash \mathtt{S}_1{\times}\mathtt{S}_2 \mathrel{<:} \mathtt{T}_1{\times}\mathtt{T}_2} \qquad \text{(S-Prod)}$$

$$\frac{\Gamma \vdash \mathtt{T}_2 \mathrel{<:} \mathtt{T}_3 \qquad \Gamma \vdash \mathtt{T}_1 \in *}{\Gamma \vdash\ \mathtt{!T}_1{\times}\mathtt{T}_2 \mathrel{<:} \mathtt{!T}_1{\times}\mathtt{T}_3} \qquad \text{(S-Upd)}$$

The subtyping rules for the type destructors are reminiscent of the generation lemma for their active counterparts: `.1` and `.2` are covariant, while `EBody` and `RBody` are invariant in their first (substitutive) arguments and covariant in their second arguments. Finally, subtyping extends η-equality.

$$\frac{\Gamma \vdash \mathtt{S} \mathrel{<:} \mathtt{T} \qquad \Gamma \vdash \mathtt{S.i} \in * \qquad \Gamma \vdash \mathtt{T.i} \in *}{\Gamma \vdash \mathtt{S.i} \mathrel{<:} \mathtt{T.i}} \qquad \text{(S-Proj)}$$

$$\frac{\Gamma \vdash \mathtt{S} \ll \mathtt{Some(X)K}_1 \qquad \Gamma \vdash \mathtt{T} \ll \mathtt{Some(X)K}_2 \qquad \Gamma \vdash \mathtt{U} \in * \qquad \Gamma \vdash \mathtt{S} \mathrel{<:} \mathtt{T}}{\Gamma \vdash \mathtt{EBody(U,S)} \mathrel{<:} \mathtt{EBody(U,T)}} \qquad \text{(S-EBody)}$$

$$\frac{\Gamma \vdash \mathtt{S} \ll \mathtt{Rec(X)S}_1 \qquad \Gamma \vdash \mathtt{T} \ll \mathtt{Rec(X)T}_1 \qquad \Gamma \vdash \mathtt{U} \in * \qquad \Gamma \vdash \mathtt{S} \mathrel{<:} \mathtt{T}}{\Gamma \vdash \mathtt{RBody(U,S)} \mathrel{<:} \mathtt{RBody(U,T)}} \qquad \text{(S-RBody)}$$

$$\frac{\Gamma \vdash \mathtt{S} =_\eta \mathtt{T} \qquad \Gamma \vdash \mathtt{S} \in *}{\Gamma \vdash \mathtt{S} \mathrel{<:} \mathtt{T}} \qquad \text{(S-Conv)}$$

The subtyping rule S-RBody is the "greatest common denominator" of the inversions of S-Rec and S-Refl which both can generate subtypings between recursive types.

Notice that the destructors occurring in these rules may be defined ones (i.e., they may be applied to active types), so, for example, the following is a valid derivation:

$$\dfrac{\dfrac{}{\mathtt{Y}{:}{*},\ \mathtt{X}{<:}\mathtt{Y}{\times}\mathtt{Y} \vdash \mathtt{X} \mathrel{<:} \mathtt{Y}{\times}\mathtt{Y}}\ \text{(S-Var)}}{\mathtt{Y}{:}{*},\ \mathtt{X}{<:}\mathtt{Y}{\times}\mathtt{Y} \vdash \mathtt{X.1} \mathrel{<:} \mathtt{Y}}\ \text{(S-Proj)}$$

The following property of kinding and subtyping fulfills the promise made in the introduction that every subtype of a product is a product, etc.

3.3.1 Theorem [Kinding is complete]:

1. If $\Gamma \vdash \mathtt{S} \mathrel{<:} \mathtt{T}_1{\times}\mathtt{T}_2$, then $\Gamma \vdash \mathtt{S} \ll \mathtt{K}_1{\times}\mathtt{K}_2$ for some $\mathtt{K}_1$ and $\mathtt{K}_2$.
2. If $\Gamma \vdash \mathtt{S} \mathrel{<:} \mathtt{!T}_1{\times}\mathtt{T}_2$, then $\Gamma \vdash \mathtt{S} \ll \mathtt{!T}_1'{\times}\mathtt{K}_2$ for some $\mathtt{T}_1'$ and $\mathtt{K}_1$ with $\Gamma \vdash \mathtt{T}_1 =_\eta \mathtt{T}_1'$.
3. If $\Gamma \vdash \mathtt{S} \mathrel{<:} \mathtt{Some(X)T}$, then $\Gamma \vdash \mathtt{S} \ll \mathtt{Some(X)K}$ for some $\mathtt{K}$.
4. If $\Gamma \vdash \mathtt{S} \mathrel{<:} \mathtt{Rec(X)T}$, then $\Gamma \vdash \mathtt{S} \ll \mathtt{Rec(X)K}$ for some $\mathtt{K}$.

We defer the proof until Section 5.3.

3.4 Typing

At the level of typing, $F_{\leq}^{TD}$ is standard. For example, we have the usual rule for forming existential packages (since our existentials are unbounded, we extend the context with the parameter binding $\mathtt{X{:}*}$):

$$\frac{\Gamma \vdash \mathtt{e} \in \mathtt{Some(X)T} \qquad \Gamma, \mathtt{X{:}*}, \mathtt{y{:}T} \vdash \mathtt{b} \in \mathtt{B} \qquad \mathtt{X} \notin FV(\mathtt{B})}{\Gamma \vdash \mathtt{open\ e\ as\ [X,y]\ in\ b} \in \mathtt{B}} \tag{T-Open}$$

The corresponding rule for `pack` is:

$$\frac{\Gamma \vdash \mathtt{E} =_\eta \mathtt{Some(X)T} \qquad \Gamma \vdash \mathtt{e} \in [\mathtt{S}/\mathtt{X}]\mathtt{T}}{\Gamma \vdash \mathtt{pack\ [S,e]\ as\ E} \in \mathtt{E}} \tag{T-Pack}$$

The `fold` and `unfold` constructors are treated as follows:

$$\frac{\Gamma \vdash \mathtt{R} =_\eta \mathtt{Rec(X)T}}{\Gamma \vdash \mathtt{unfold\ [R]} \in \mathtt{R} \ \rightarrow\ [\mathtt{R}/\mathtt{X}]\mathtt{T}} \tag{T-Unfold}$$

$$\frac{\Gamma \vdash \mathtt{R} =_\eta \mathtt{Rec(X)T}}{\Gamma \vdash \mathtt{fold\ [R]} \in [\mathtt{R}/\mathtt{X}]\mathtt{T} \ \rightarrow\ \mathtt{R}} \tag{T-Fold}$$

The typing relation also includes the usual rule of subsumption:

$$\frac{\Gamma \vdash \mathtt{e} \in \mathtt{S} \qquad \Gamma \vdash \mathtt{S} <: \mathtt{T}}{\Gamma \vdash \mathtt{e} \in \mathtt{T}} \tag{T-Subsumption}$$

As an example of the use of these rules, note that Abadi and Cardelli's "structural rule" for `unfold` expressions [AC96]

$$\frac{\Gamma \vdash \mathtt{e} \in \mathtt{R} <: \mathtt{Rec(X)T}}{\Gamma \vdash \mathtt{unfold\ [R]\ \ e} \in [\mathtt{R}/\mathtt{X}]\mathtt{T}}$$

is derivable in $F_{\leq}^{TD}$. If $\mathtt{R} <: \mathtt{Rec(X)T}$ then $\mathtt{R} \ll \mathtt{Rec(X)K}$ by Theorem 3.3.1, so $\mathtt{R} =_\eta \mathtt{Rec(X)\ RBody(X,R)}$. Hence, if $\mathtt{e} \in \mathtt{R}$, then $\mathtt{e} \in \mathtt{Rec(X)\ RBody(X,R)}$ by S-Conv and T-Subsumption, so

$$\begin{array}{lll} \mathtt{unfold\ [R]\ e} & \in & \mathtt{RBody(R,\ R)} \\ & <: & \mathtt{RBody(R,\ Rec(X)T)} \\ & \text{i.e.} & [\mathtt{R}/\mathtt{X}]\mathtt{T}. \end{array}$$

4 Examples

We now show how to extend the simple examples discussed so far to full-scale object encodings. We treat both of the well-known "simple encodings" of objects (cf. [BCP97])—one using recursive types to hide the types of instance variables and one using existential types. All the examples have been mechanically checked by our prototype implementation.

4.1 Record Syntax

To make the examples easier to read, we extend the system $F_{\leq}^{TD}$ with conventional record notation. We introduce the following new syntactic forms:

$$\begin{array}{lll} \mathtt{A} & ::= & \mathtt{...} \\ & & \{[\mathtt{!}]\mathtt{l}_1\mathtt{:T}_1\mathtt{;}\ \ \mathtt{...;}\ \ [\mathtt{!}]\mathtt{l}_n\mathtt{:T}_n\} \quad \text{record type} \end{array}$$

```
e  ::=  ...
        {[!]l1=e1; ...; [!]ln=en}   record value
        e.l                         projection of non-updatable field l
        e..l                        projection of updatable field l
        e1 with l:=e2               update of updatable field l in e1
```

Each field in a record is either updatable or non-updatable. For non-updatable fields, we provide the usual projection operator `e.l`. For updatable fields, we provide both projection (written `e..l`, since its encoding below is different than that of ordinary projection) and update: if `r` is a record with a updatable field `l` and `v` is a value of the appropriate type, then `r with l:=v` denotes a new record that coincides with `r` except at `l`, where its value is `v`.

These syntactic forms can all be encoded in our calculus, using a slight extension of a now-standard technique due to Cardelli [Car92]. The idea is quite simple, so we explain it informally rather than writing out a translation in full.

First, we choose some enumeration of all the labels that can appear in records. Now, an ordinary record type (with only non-updatable fields) is encoded in terms of the ordinary product type and `Top` in the usual way: by sorting its fields into the order determined by the chosen enumeration, inserting instances of `Top` for labels that do not appear in the given record type, placing a `Top` at the end, and finally dropping the labels. For example, if we take labels in alphabetical order (`a`, `b`, `c`, etc.), then the record type `{b:String}` is encoded as `(Top×String×Top)`, while `{d:Int,b:String}` is encoded as `(Top×String×Top×Int×Top)`. The encodings of record values and projection follow the same lines: `{b="red"}` is encoded as `(top, ("red", top))`, where `top` is an arbitrary value; `r.b` is encoded as `r.2.1`.

An updatable field of type `T` is encoded by placing the pair `(!T×Top)` in the appropriate position, rather than just `T`. For example, the updatable record `{!b:String}` is encoded as `(Top×(!String×Top)×Top)`. Field values and projection are encoded in the obvious way: the record creation expression `{!b="red"}` becomes `(top, ((!"red",top), top))`, and `r..b` becomes `r.2.1.1`. Finally, `with`-expressions are encoded by building a new record from the pieces of the original: for example, the update expression `r with b:="green"` becomes `(r.1, ((!"green", r.2.1.2), r.2.2))`. It is easy to verify that these encodings satisfy the expected typing and subtyping rules.

In a future version of $F^{TD}_{\leq}$, we would like to include a subtyping rule of the form $!T_1 \times T_2$ <: $T_1 \times T_2$. This would give us a neater encoding of records, using updatable products directly for updatable fields. For example, `{d:Int,!b:String}` would become `(Top×!String×Top×Int×Top)`. This encoding is not adequate in the present system because it disallows adding updatable fields while subtyping. For example, `{!a:Int,d:Int,!b:String}` would not be a subtype of `{d:Int,!b:String}`.

4.2 Recursive Objects

We now present a simple "objects as recursive records" encoding. The idea of the encoding is standard (cf. [BCP97] for details and references). What is interesting is the way the destructor for recursive types is used to achieve "polymorphic unfolding" in a style reminiscent of [ACV96, AC96], rather than using higher-order quantification [PT94, HP95b] or matching [BPF97, AC95] to give sufficiently refined types to the message-sending operators. The whole encoding can thus be carried out in a second-order setting.

Our running example will be the usual "functional reference cell," a simple object with three methods: `get`, `set`, and `bump`. The type of cell objects under this encoding is a recursively defined record type with three fields giving the result types of the three methods. For brevity, we'll use `Cell` in this section as an abbreviation for this type:

```
Cell  =  Rec(X) {get:Int; set:Int→X; bump:X}
```

An object with this type can be created as follows:

```
val o =
  let create =
    fix [Int→Cell]
      (fun(c:Int→Cell)
        fun(s:Int)
```

```
            fold [Cell]
              {get = s;
               set = fun(i:Int) c(i);
               bump = c(succ s)})
    in
      create(0)
    :: Cell
```

That is, we build a cell object by defining a recursive function `create` that, given an integer (representing the state of the cell) returns a record of method results, where the `set` and `bump` results are calculated by calling `create` with an appropriately updated value for the state. This function is applied to the initial state `0` to create the cell object `o`. The recursive definition of `create` uses the value-level polymorphic fixed-point operator `fix`, which can be defined in terms of recursive types [AC93].

The interesting part of the example is the typing of functions that manipulate objects by sending them messages (i.e., by unfolding the outer recursive type once and projecting one of the fields). For example, the following function sends the `get` message to an arbitrary object whose type refines `Cell`:

```
val sendget =
  fun(Y<:Cell) fun(or:Y)
    (unfold [Y] or).get
  :: All(Y<:Cell) Y→Int
```

The `sendget` function can be typed without using the special features of $F_{\leq}^{TD}$. But the analogous `sendbump` function

```
val sendbump =
  fun(Y<:Cell) fun(or:Y)
    (unfold [Y] or).bump
  :: All(Y<:Cell) Y→Y
```

uses type destructors in an essential way. Its type can be calculated as follows:

`Y`	<:	`Cell`	given
	=	`Rec(X) {get:Int; set:Int→X; bump:X}`	by definition
`Y`	$\ll$	`Rec(X) K`	by Theorem 3.3.1
`Y`	$=_\eta$	`Rec(Z)RBody(Z,Y)`	by Eta-Rec
`unfold [Y]`	$\in$	`Y → [Y/Z]RBody(Z,Y)`	by T-Unfold
	=	`Y → RBody(Y,Y)`	by defn of substitution
`unfold [Y] or`	$\in$	`RBody(Y,Y)`	by application
	<:	`[Cell/W]RBody(Y,W)`	by S-RBody
	=	`[Cell/W][Y/X]{get:Int; set:Int→X; bump:X}`	by defn of substitution
	=	`{get:Int; set:Int→Y; bump:Y}`	by defn of substitution
`(unfold [Y] or).bump`	$\in$	`Y`	by projection
`sendbump`	$\in$	`All(Y<:Cell) Y→Y`	by abstraction.

4.3 Existential Objects

The "simple existential" encoding of objects [PT94, HP95b, etc.] can also be formulated in $F_{\leq}^{TD}$. Again, the presence of type destructors allows functions manipulating objects (`sendbump`, etc.) to be written in a direct and intuitive way.

For this encoding, we keep the same interface for the cell methods, but change the type of objects so that the "state component" of an object is made visible but its type is hidden with an existential quantifier:

```
Cell  =  Some(X) !X × (X → {get:Int; set:Int→X; bump:X})
```

That is, a cell object is a pair of a state of type `X` and a collection of methods mapping `X` to the result types specified by `CellI`, with the type of the state existentially quantified. Note that the state component is updatable.

Functions that manipulate objects by sending them messages are slightly more complicated here than in the recursive records model (where an object simply *was* a record of the results of its methods). For example, to send the `get` message to a cell object

```
val sendget =
  fun(Y<:Cell) fun(oe:Y)
    open oe as [Z,body] in
       (body.2 body.1).get
  :: All(Y<:Cell) Y→Int
```

we must first `open` it, binding a type variable `Y` to its hidden state type and a variable `body` to its state and methods. The methods (`body.2`) are then applied to the state (`body.1`), yielding a record of results, from which the `get` component is selected. As for the recursive record encoding, the typing of `sendget` is just as in $F_{\leq}$.

To send the `bump` message, we begin as for `get`, applying the methods to the state and projecting out the `bump` component; but this yields just a fresh state (of type `Y`), not a whole object. To obtain an object, we must repackage this state with the original methods and hide the type of the state by wrapping the whole in a new existential package:

```
val sendbump =
  fun(Y<:Cell) fun(oe:Y)
    open oe as [Z,body] in
       pack [Z, !(body.2 body.1).bump, body.2]
       as Y
  :: All(Y<:Cell) Y→Y
```

To check that `sendbump` has the claimed type, calculate as follows. First, as in the previous section:

`Y`	<:	`Cell`	given
	=	`Some(X) !X × X→{get:Int;set:Int→X;bump:X}`	by definition
`Y`	$\ll$	`Some(X) K`	by Theorem 3.3.1
`Y`	$=_\eta$	`Some(Z) EBody(Z,Y)`	by ETA-SOME.

So (by T-OPEN), in the body of the `open` expression, the bindings of `Z` and `body` are:

`Z:*`
`body` ∈ `EBody(Z,Y)`.

Now,

`EBody(Z,Y)`	<:	`EBody(Z, Some(X)!X×X→{get:Int;set:Int→X;bump:X})`	by S-EBODY
	=	`!Z × Z→{get:Int;set:Int→Z;bump:Z}`	defn of substitution
`EBody(Z,Y)`	$\ll$	`!Z′ × K₂` (with `Z′` $=_\eta$ `Z`)	Theorem 3.3.1
`EBody(Z,Y)`	$=_\eta$	`!Z′ × EBody(Z,Y).2`	ETA-UPD
	$=_\eta$	`!Z × EBody(Z,Y).2`	$=_\eta$ is a congruence,

so

`body.2` ∈ `EBody(Z,Y).2`

by projection. Moreover,

`EBody(Z,Y).2`	<:	`EBody(Z,Cell).2`	by S-EBODY and S-PROD
	=	`(!Z × Z→{get:Int;set:Int→Z;bump:Z}).2`	by defn of substitution
	=	`Z→{get:Int;set:Int→Z;bump:Z}`	by defn of substitution,

so

`(body.2 body.1).bump` ∈ `Z`

by projection. Thus,

$$\begin{array}{rcl} \texttt{!(body.2 body.1).bump, body.2} & \in & \texttt{!Z} \times \texttt{EBody(Z,Y).2} \\ & =_\eta & \texttt{EBody(Z,Y),} \end{array}$$

and hence

```
pack [Z, !(body.2 body.1).bump, body.2] as Y
```

has type Y by T-PACK, from which the claimed typing of sendbump follows by abstraction.

As before, creating a cell object with appropriate behavior is straightforward. We simply pair the initial state together with a method function and wrap the two as an existential package:

```
val o =
  pack [Int,
        !0,
        fun(s:Int)
          {get = s;
           set = fun(i:Int) i;
           bump = succ s}
       ] as Cell
  :: Cell
```

Of course, not only objects but also classes can be encoded in this framework. The power of type destructors is not needed for this encoding, but (as has been remarked elsewhere [HP95a, Pol96, etc.]) the presence of updatable record types does eliminate quite a bit of distracting boilerplate (the get and put functions of [PT94]).

5 Metatheory

We now develop basic metatheoretic properties of $F_{\leq}^{TD}$.

5.1 Kinding

Kinding is defined by a syntax-directed procedure and so is decidable (cf. Proposition 5.5.1). For what follows, we need some additional facts about how kinding behaves with respect to substitution for parameters.

5.1.1 Lemma [Kinding and parameter substitution]: Suppose that $\Gamma, \mathtt{X{:}*}, \Delta \vdash \mathtt{T} \ll \mathtt{B}$ and $\Gamma \vdash \mathtt{S} \ll \mathtt{A}$, and that $[\mathtt{S}/\mathtt{X}]\Delta$ is defined. Then $[\mathtt{S}/\mathtt{X}]\mathtt{T}$ and $[\mathtt{S},\mathtt{A}/\mathtt{X}]\mathtt{B}$ are defined and $\Gamma, [\mathtt{S}/\mathtt{X}]\Delta \vdash [\mathtt{S}/\mathtt{X}]\mathtt{T} \ll [\mathtt{S},\mathtt{A}/\mathtt{X}]\mathtt{B}$.

Proof: That $[\mathtt{S}/\mathtt{X}]\mathtt{T}$ and $[\mathtt{S},\mathtt{A}/\mathtt{X}]\mathtt{B}$ are defined is obvious: X cannot appear inside a destructing context such as X.1 or EBody(T,X), so the substitution is entirely structural. Similarly, if Δ is well-kinded, then $[\mathtt{S}/\mathtt{X}]\Delta$ will be defined. The second part goes by induction on a derivation of $\Gamma, \mathtt{X{:}*}, \Delta \vdash \mathtt{T} \ll \mathtt{B}$.

Case: $\mathtt{T} = \mathtt{Some(Y)T}_1$

Then $\mathtt{B} = \mathtt{Some(Y)B}_1$ with $\mathtt{Y{:}*} \vdash \mathtt{T}_1 \ll \mathtt{B}_1$. The induction hypothesis gives $[\mathtt{S}/\mathtt{X}]\mathtt{T}_1 \ll [\mathtt{S},\mathtt{A}/\mathtt{X}]\mathtt{B}_1$, and thus $[\mathtt{S}/\mathtt{X}]\mathtt{T} \ll \mathtt{Some(Y)}[\mathtt{S},\mathtt{A}/\mathtt{X}]\mathtt{B}_1 = [\mathtt{S},\mathtt{A}/\mathtt{X}]\mathtt{B}$.

Case: $\mathtt{T} = \mathtt{EBody(T}_1\mathtt{,N)}$

Then $\mathtt{B} = [\mathtt{T}_1,\mathtt{B}_1/\mathtt{Y}]\mathtt{B}_2$ with $\mathtt{T}_1 \ll \mathtt{B}_1$ and $\mathtt{N} \ll \mathtt{Some(Y)B}_2$. The induction hypothesis gives $[\mathtt{S}/\mathtt{X}]\mathtt{T}_1 \ll [\mathtt{S},\mathtt{A}/\mathtt{X}]\mathtt{B}_1$ and $[\mathtt{S}/\mathtt{X}]\mathtt{N} \ll \mathtt{Some(Y)}[\mathtt{S},\mathtt{A}/\mathtt{X}]\mathtt{B}_2$. Thus, $[\mathtt{S}/\mathtt{X}]\mathtt{T} = \mathtt{EBody(}[\mathtt{S}/\mathtt{X}]\mathtt{T}_1, [\mathtt{S}/\mathtt{X}]\mathtt{N}) \ll [[\mathtt{S}/\mathtt{X}]\mathtt{T}_1, [\mathtt{S},\mathtt{A}/\mathtt{X}]\mathtt{B}_1/\mathtt{Y}][\mathtt{S},\mathtt{A}/\mathtt{X}]\mathtt{B}_2 = [\mathtt{S},\mathtt{A}/\mathtt{X}][\mathtt{T}_1,\mathtt{B}_1/\mathtt{Y}]\mathtt{B}_2 = [\mathtt{S},\mathtt{A}/\mathtt{X}]\mathtt{B}$.

Other cases:
Similar. ■

5.1.2 Corollary [Soundness of derived kinding rules for destructors]: Generalizations of the kinding rules for K-FST through K-RBODY where active types are allowed to appear in place of neutral types are derivable; for example, if $\Gamma \vdash \mathtt{A} \ll \mathtt{Rec(X)K}$ and $\Gamma \vdash \mathtt{T} \ll \mathtt{L}$, then $\Gamma \vdash \mathtt{RBody(T,A)} \ll [\mathtt{T},\mathtt{L}/\mathtt{X}]\mathtt{K}$.

Proof: Immediate from the substitution lemma and generation of $\ll$. For example, suppose $\Gamma \vdash \mathtt{A} \ll \mathtt{Rec(X)K}$. Then, by generation of kinding, we must have $\mathtt{A} = \mathtt{Rec(X)T}$ and $\Gamma,\mathtt{X{:}*} \vdash \mathtt{T} \ll \mathtt{K}$. If, in addition, $\Gamma \vdash \mathtt{U} \ll \mathtt{L}$, then $\mathtt{RBody(U,A)}$ is definitionally equal to $\mathtt{[U/X]T}$. From Lemma 5.1.1 we get $\Gamma \vdash \mathtt{[U/X]T} \ll \mathtt{[U,L/X]K}$, hence $\Gamma \vdash \mathtt{RBody(U,A)} \ll \mathtt{[U,L/X]K}$, from which the desired conclusion follows. ■

5.1.3 Lemma [Kinding is idempotent]: If $\mathtt{S} \ll \mathtt{K}$ then $\Gamma \vdash \mathtt{K} \ll \mathtt{K}$.

For the proof, we need the following sub-lemma:

5.1.4 Lemma: If $\mathtt{X{:}*} \vdash \mathtt{L} \ll \mathtt{L}$ and $\mathtt{T} \ll \mathtt{M}$, then $[\mathtt{T},\mathtt{M}/\mathtt{X}]\mathtt{L} \ll [\mathtt{T},\mathtt{M}/\mathtt{X}]\mathtt{L}$.

Proof of 5.1.4: By induction on $\mathtt{L}$. If $\mathtt{L} = \mathtt{!S}{\times}\mathtt{L}_2$ then $\mathtt{S} \ll \mathtt{L}_1$ and $\mathtt{L}_2 \ll \mathtt{L}_2$. Lemma 5.1.1 guarantees that $[\mathtt{T}/\mathtt{X}]\mathtt{S} \in *$ (that is, $[\mathtt{T}/\mathtt{X}]\mathtt{S}$ is well kinded), and the result follows by the induction hypothesis and K-UPD. If $\mathtt{L} = \mathtt{Some(Y)L}_1$ then $\mathtt{Y{:}*} \vdash \mathtt{L}_1 \ll \mathtt{L}_1$, and thus $\mathtt{Y{:}*} \vdash [\mathtt{T},\mathtt{M}/\mathtt{X}]\mathtt{L}_1 \ll [\mathtt{T},\mathtt{M}/\mathtt{X}]\mathtt{L}_1$, and the result follows by K-SOME. Similarly for the other type formers. ■

Proof of 5.1.3: By induction on the derivation of $\mathtt{S} \ll \mathtt{K}$. All cases except K-EBODY and K-RBODY are straightforward. Suppose, therefore, that $\mathtt{S} = \mathtt{RBody(T,N)}$ and $\mathtt{T} \ll \mathtt{K}_1$ and $\mathtt{N} \ll \mathtt{Rec(X)K}_2$. The induction hypothesis gives $\mathtt{K}_1 \ll \mathtt{K}_1$ and $\mathtt{X{:}*} \vdash \mathtt{K}_2 \ll \mathtt{K}_2$. Lemma 5.1.4 gives $[\mathtt{T},\mathtt{K}_1/\mathtt{X}]\mathtt{K}_2 \ll [\mathtt{T},\mathtt{K}_1/\mathtt{X}]\mathtt{K}_2$, which is the required conclusion. The argument for K-EBODY is similar. ■

5.2 Algorithmic Eta-Conversion

In order to decide eta-conversion we introduce the following syntax-directed rules:

$$\frac{\Gamma \vdash \mathtt{S} \in *}{\Gamma \vdash \mathtt{S} =_\eta \mathtt{S}} \qquad \text{(ETA-A-REFL)}$$

$$\frac{\Gamma \vdash \mathtt{S}_1 =_\eta \mathtt{T}_1 \qquad \Gamma \vdash \mathtt{S}_2 =_\eta \mathtt{T}_2}{\Gamma \vdash \mathtt{S}_1 \diamondsuit \mathtt{S}_2 =_\eta \mathtt{T}_1 \diamondsuit \mathtt{T}_2} \qquad \text{(ETA-A-ANY)}$$

$$\frac{\Gamma, \mathtt{X{<:}S}_1 \vdash \mathtt{S}_2 =_\eta \mathtt{T}_2}{\Gamma \vdash \mathtt{All(X{<:}S}_1\mathtt{)S}_2 =_\eta \mathtt{All(X{<:}S}_1\mathtt{)T}_2} \qquad \text{(ETA-A-ALL)}$$

$$\frac{\Gamma, \mathtt{X{:}*} \vdash \mathtt{S}_2 =_\eta \mathtt{T}_2}{\Gamma \vdash \mathtt{Some(X)S}_2 =_\eta \mathtt{Some(X)T}_2} \qquad \text{(ETA-A-SOME)}$$

$$\frac{\Gamma, \mathtt{X{:}*} \vdash \mathtt{S} =_\eta \mathtt{T}}{\Gamma \vdash \mathtt{Rec(X)S} =_\eta \mathtt{Rec(X)T}} \qquad \text{(ETA-A-REC)}$$

$$\frac{\Gamma \vdash \mathtt{N} \ll \mathtt{K}_1{\times}\mathtt{K}_2 \qquad \Gamma \vdash \mathtt{N.1} =_\eta \mathtt{T}_1 \qquad \Gamma \vdash \mathtt{N.2} =_\eta \mathtt{T}_2}{\Gamma \vdash \mathtt{N} =_\eta \mathtt{T}_1{\times}\mathtt{T}_2} \qquad \text{(ETA-AL-PROD)}$$

$$\frac{\Gamma \vdash \mathtt{N} \ll \mathtt{K}_1{\times}\mathtt{K}_2 \qquad \Gamma \vdash \mathtt{N.1} =_\eta \mathtt{T}_1 \qquad \Gamma \vdash \mathtt{N.2} =_\eta \mathtt{T}_2}{\Gamma \vdash \mathtt{T}_1{\times}\mathtt{T}_2 =_\eta \mathtt{N}} \qquad \text{(ETA-AR-PROD)}$$

$$\frac{\Gamma \vdash \mathtt{N} \ll \mathtt{!S_1{\times}K_2} \qquad \Gamma \vdash \mathtt{S_1} =_\eta \mathtt{T_1} \qquad \Gamma \vdash \mathtt{N.2} =_\eta \mathtt{T_2}}{\Gamma \vdash \mathtt{N} =_\eta \mathtt{!T_1{\times}T_2}} \quad \text{(Eta-AL-Upd)}$$

$$\frac{\Gamma \vdash \mathtt{N} \ll \mathtt{!S_1{\times}K_2} \qquad \Gamma \vdash \mathtt{S_1} =_\eta \mathtt{T_1} \qquad \Gamma \vdash \mathtt{N.2} =_\eta \mathtt{T_2}}{\Gamma \vdash \mathtt{!T_1{\times}T_2} =_\eta \mathtt{N}} \quad \text{(Eta-AR-Upd)}$$

$$\frac{\Gamma \vdash \mathtt{N} \ll \mathtt{Some(X)K} \qquad \Gamma, \mathtt{X{:}*} \vdash \mathtt{EBody(X,N)} =_\eta \mathtt{T}}{\Gamma \vdash \mathtt{N} =_\eta \mathtt{Some(X)T}} \quad \text{(Eta-AL-Some)}$$

$$\frac{\Gamma \vdash \mathtt{N} \ll \mathtt{Some(X)K} \qquad \Gamma, \mathtt{X{:}*} \vdash \mathtt{EBody(X,N)} =_\eta \mathtt{T}}{\Gamma \vdash \mathtt{Some(X)T} =_\eta \mathtt{N}} \quad \text{(Eta-AR-Some)}$$

$$\frac{\Gamma \vdash \mathtt{N} \ll \mathtt{Rec(X)K} \qquad \Gamma, \mathtt{X{:}*} \vdash \mathtt{RBody(X,N)} =_\eta \mathtt{T}}{\Gamma \vdash \mathtt{N} =_\eta \mathtt{Rec(X)T}} \quad \text{(Eta-AL-Rec)}$$

$$\frac{\Gamma \vdash \mathtt{N} \ll \mathtt{Rec(X)K} \qquad \Gamma, \mathtt{X{:}*} \vdash \mathtt{RBody(X,N)} =_\eta \mathtt{T}}{\Gamma \vdash \mathtt{Rec(X)T} =_\eta \mathtt{N}} \quad \text{(Eta-AR-Rec)}$$

The Eta-AL-... and Eta-AR-... rules will be referred to collectively as *LR-rules*.

Each of these rules is easily derived from the declarative eta-conversion rules given in Section 3.2. Moreover, most of the definition in Section 3.2 is mirrored directly here: Eta-A-Refl is an explicit symmetry rule, while Eta-A-Any through Eta-A-Rec give explicit congruence rules for all the type constructors. (If we had defined the original eta-converstion relation to be a full congruence—allowing eta-conversion inside bounds of quantifiers and substitutive arguments of `EBody` and `RBody`—we would need to introduce congruence rules for destructors here as well.) The remaining algorithmic rules correspond to special uses of the original declarative rules, where an instance of transitivity has been "pushed into" each premise. Our main job in this section will be to show that the algorithmic presentation itself defines a transitive relation.

When we need to distinguish the alorithmic from the ordinary eta-conversion relation, we will write $\Gamma \vdash^a \mathtt{S} =_\eta \mathtt{T}$ for algorithmic derivations.

5.2.1 Definition: The *size* of a algorithmic eta-conversion derivation is the number of Eta- rules it contains. (Kinding premises do not count toward size.)

5.2.2 Proposition [Eta and parameter substitution]: If $\mathcal{D} :: \Gamma, \mathtt{X{:}*}, \Delta \vdash^a \mathtt{S} =_\eta \mathtt{T}$ and $\Gamma \vdash \mathtt{V} \in *$ and $[\mathtt{V/X}]\Delta$ is defined, then $\Gamma, [\mathtt{V/X}]\Delta \vdash^a [\mathtt{V/X}]\mathtt{S} =_\eta [\mathtt{V/X}]\mathtt{T}$ by a derivation not larger than $\mathcal{D}$.

Proof: Straightforward induction on derivations, using Lemma 5.1.1 for the LR-rules and Eta-A-Refl. Note that Eta-A-Refl applies to arbitrary types not only neutral ones. ■

5.2.3 Proposition [Eta-congruence for destructors]: Let $\mathtt{Z(Y)}$ be `Y.1`, `Y.2`, `EBody(P,Y)`, or `RBody(P,Y)`. If $\mathcal{D} :: \Gamma \vdash^a \mathtt{S} =_\eta \mathtt{T}$ and $\mathtt{Z(S)}$ and $\mathtt{Z(T)}$ are well kinded, then $\Gamma \vdash^a \mathtt{Z(S)} =_\eta \mathtt{Z(T)}$ by a derivation not larger than $\mathcal{D}$.

Proof: If $\mathcal{D}$ is an instance of Eta-A-Refl, then the result is an instance of reflexivity. If $\mathcal{D}$ ends in one of Eta-A-Any...Eta-AL-Rec, then the result can be obtained from one of the immediate premises of $\mathcal{D}$ using Proposition 5.2.2. ■

5.2.4 Lemma: $=_\eta$ is symmetric. Moreover, if $\mathcal{D} :: \Gamma \vdash \mathtt{S} =_\eta \mathtt{T}$, then there exists a derivation $\mathcal{D}' :: \Gamma \vdash \mathtt{T} =_\eta \mathtt{S}$ of the same size as $\mathcal{D}$.

Proof: Easy induction on $\mathcal{D}$. ■

5.2.5 Proposition: $=_\eta$ is transitive.

Proof: Transitivity follows by simultaneous induction on derivations. Suppose, for example, that we have proved $\mathtt{N} =_\eta \mathtt{Some(X)T}$ from $\mathtt{N} \ll \mathtt{Some(X)K_2}$ and $\mathtt{EBody(X,N)} =_\eta \mathtt{T}$ using ETA-AL-SOME, and that we have $\mathtt{Some(X)T} =_\eta \mathtt{Some(X)U}$ from $\mathtt{X{:}*} \vdash^a \mathtt{T} =_\eta \mathtt{U}$ by ETA-A-SOME. The induction hypothesis then yields $\mathtt{EBody(X,N)} =_\eta \mathtt{U}$, hence $\mathtt{N} =_\eta \mathtt{Some(X)U}$ by ETA-AL-SOME. If $\mathtt{Some(X)T} =_\eta \mathtt{N}'$ has been derived by ETA-AR-SOME, then the induction hypothesis yields $\mathtt{EBody(X,N)} =_\eta \mathtt{EBody(X,N')}$, hence $\mathtt{N} = \mathtt{N}'$, since no algorithmic eta-rule except reflexivity applies to neutral types. ∎

We have thus established:

5.2.6 Theorem: $\Gamma \vdash \mathtt{S} =_\eta \mathtt{T}$ under the declarative definition iff $\Gamma \vdash^a \mathtt{S} =_\eta \mathtt{T}$ can be proved using the algorithmic rules.

5.3 Algorithmic Subtyping

In this section, we define an algorithmic subtyping judgement $\Gamma \vdash \mathtt{S} <: \mathtt{T}$, which gives rise to a syntax-directed decision procedure for subtyping. For the whole of Section 5.3, the symbol $\vdash$ and the words "derive," "derivable," etc. refer to algorithmic derivations (for both subtyping and eta-conversion).

Like the algorithmic eta-conversion relation defined in the previous section, algorithmic subtyping does not explicitly contain a transitivity rule; instead we have a promotion rule which, roughly speaking, allows us to replace the head variable of a neutral type by its upper bound.

5.3.1 Definition: Let N be a well-kinded neutral type in context Γ. The promotion $\Gamma(\mathtt{N})$ of N is given by

$$
\begin{array}{lcll}
\Gamma(\mathtt{X}) & = & \mathtt{T} & \text{if } \mathtt{X<:T} \in \Gamma \\
\Gamma(\mathtt{X}) & = & * & \text{if } \mathtt{X{:}*} \in \Gamma \\
\Gamma(\mathtt{N.1}) & = & \Gamma(\mathtt{N})\mathtt{.1} & \\
\Gamma(\mathtt{N.2}) & = & \Gamma(\mathtt{N})\mathtt{.2} & \\
\Gamma(\mathtt{EBody(T,N)}) & = & \mathtt{EBody(T,}\Gamma(\mathtt{N})\mathtt{)} & \\
\Gamma(\mathtt{RBody(T,N)}) & = & \mathtt{RBody(T,}\Gamma(\mathtt{N})\mathtt{)} &
\end{array}
$$

For example, if $\Gamma = \mathtt{X<:Some(X)X{\times}Top}$ and $\mathtt{N} = \mathtt{EBody(Int,X).1}$, then $\Gamma(\mathtt{N}) = \mathtt{Int}$. Notice that, by Lemma 5.1.1, the promotion of N is always defined, since all the substitutions involved are parameter substitutions.

The promotion rule now takes the form

$$
\frac{\Gamma(\mathtt{N}) \neq * \qquad \Gamma \vdash \Gamma(\mathtt{N}) <: \mathtt{T} \qquad \Gamma \vdash \mathtt{N} \ll \mathtt{K} \qquad \mathtt{T} \text{ neutral or } \mathtt{K} = \mathtt{Top}}{\Gamma \vdash \mathtt{N} <: \mathtt{T}} \quad \text{(SA-PROMOTE)}
$$

where the final premise ensures that SA-PROMOTE can be applied only if no other rule applies. The other algorithmic rules are as follows.

$$
\frac{\Gamma \vdash \mathtt{S} \in *}{\Gamma \vdash \mathtt{S} <: \mathtt{S}} \quad \text{(SA-REFL)}
$$

$$
\frac{\Gamma \vdash \mathtt{T} \in *}{\Gamma \vdash \mathtt{T} <: \mathtt{Top}} \quad \text{(SA-TOP)}
$$

$$
\frac{\Gamma \vdash \mathtt{S_1} <: \mathtt{T_1} \qquad \Gamma \vdash \mathtt{S_2} <: \mathtt{T_2}}{\Gamma \vdash \mathtt{S_1{\times}S_2} <: \mathtt{T_1{\times}T_2}} \quad \text{(SA-PROD)}
$$

$$
\frac{\Gamma \vdash \mathtt{S_1} =_\eta \mathtt{T_1} \qquad \Gamma \vdash \mathtt{S_2} <: \mathtt{T_2}}{\Gamma \vdash \mathtt{!S_1{\times}S_2} <: \mathtt{!T_1{\times}T_2}} \quad \text{(SA-UPD)}
$$

$$\frac{\Gamma \vdash \mathtt{T}_1 \mathrel{<:} \mathtt{S}_1 \qquad \Gamma \vdash \mathtt{S}_2 \mathrel{<:} \mathtt{T}_2}{\Gamma \vdash \mathtt{S}_1 \to \mathtt{S}_2 \mathrel{<:} \mathtt{T}_1 \to \mathtt{T}_2} \tag{SA-Arrow}$$

$$\frac{\Gamma \vdash \mathtt{S}_1 \in * \qquad \Gamma, \mathtt{X} \mathrel{<:} \mathtt{S}_1 \vdash \mathtt{S}_2 \mathrel{<:} \mathtt{T}_2}{\Gamma \vdash \mathtt{All(X}\mathrel{<:}\mathtt{S}_1\mathtt{)S}_2 \mathrel{<:} \mathtt{All(X}\mathrel{<:}\mathtt{S}_1\mathtt{)T}_2} \tag{SA-All}$$

$$\frac{\Gamma, \mathtt{X}:* \vdash \mathtt{S} \mathrel{<:} \mathtt{T}}{\Gamma \vdash \mathtt{Some(X)S} \mathrel{<:} \mathtt{Some(X)T}} \tag{SA-Some}$$

$$\frac{\Gamma, \mathtt{Y}:*, \mathtt{X} \mathrel{<:} \mathtt{Y} \vdash \mathtt{S} \mathrel{<:} \mathtt{T}}{\Gamma \vdash \mathtt{Rec(X)S} \mathrel{<:} \mathtt{Rec(Y)T}} \tag{SA-Rec}$$

$$\frac{\Gamma, \mathtt{X}:* \vdash \mathtt{S} =_\eta \mathtt{T}}{\Gamma \vdash \mathtt{Rec(X)S} \mathrel{<:} \mathtt{Rec(X)T}} \tag{SA-Rec'}$$

$$\frac{\Gamma \vdash \mathtt{N} \ll \mathtt{K}_1 \times \mathtt{K}_2 \qquad \Gamma \vdash \mathtt{N.1} \mathrel{<:} \mathtt{T}_1 \qquad \Gamma \vdash \mathtt{N.2} \mathrel{<:} \mathtt{T}_2}{\Gamma \vdash \mathtt{N} \mathrel{<:} \mathtt{T}_1 \times \mathtt{T}_2} \tag{SAL-Prod}$$

$$\frac{\Gamma \vdash \mathtt{N} \ll \mathtt{K}_1 \times \mathtt{K}_2 \qquad \Gamma \vdash \mathtt{S}_1 \mathrel{<:} \mathtt{N.1} \qquad \Gamma \vdash \mathtt{S}_2 \mathrel{<:} \mathtt{N.2}}{\Gamma \vdash \mathtt{S}_1 \times \mathtt{S}_2 \mathrel{<:} \mathtt{N}} \tag{SAR-Prod}$$

$$\frac{\Gamma \vdash \mathtt{N} \ll \mathtt{!S}_1 \times \mathtt{K}_2 \qquad \Gamma \vdash \mathtt{S}_1 =_\eta \mathtt{T}_1 \qquad \Gamma \vdash \mathtt{N.2} \mathrel{<:} \mathtt{T}_2}{\Gamma \vdash \mathtt{N} \mathrel{<:} \mathtt{!T}_1 \times \mathtt{T}_2} \tag{SAL-Upd}$$

$$\frac{\Gamma \vdash \mathtt{N} \ll \mathtt{!T}_1 \times \mathtt{K}_2 \qquad \Gamma \vdash \mathtt{S}_1 =_\eta \mathtt{T}_1 \qquad \Gamma \vdash \mathtt{S}_2 \mathrel{<:} \mathtt{N.2}}{\Gamma \vdash \mathtt{!S}_1 \times \mathtt{S}_2 \mathrel{<:} \mathtt{N}} \tag{SAR-Upd}$$

$$\frac{\Gamma \vdash \mathtt{N} \ll \mathtt{Some(X)K} \qquad \Gamma, \mathtt{X}:* \vdash \mathtt{EBody(X,N)} \mathrel{<:} \mathtt{T}}{\Gamma \vdash \mathtt{N} \mathrel{<:} \mathtt{Some(X)T}} \tag{SAL-Some}$$

$$\frac{\Gamma \vdash \mathtt{N} \ll \mathtt{Some(X)K} \qquad \Gamma, \mathtt{X}:* \vdash \mathtt{S} \mathrel{<:} \mathtt{EBody(X,N)}}{\Gamma \vdash \mathtt{Some(X)S} \mathrel{<:} \mathtt{N}} \tag{SAR-Some}$$

$$\frac{\Gamma \vdash \mathtt{N} \ll \mathtt{Rec(X)K} \qquad \Gamma, \mathtt{Y}:*, \mathtt{X} \mathrel{<:} \mathtt{Y} \vdash \mathtt{RBody(X,N)} \mathrel{<:} \mathtt{T}}{\Gamma \vdash \mathtt{N} \mathrel{<:} \mathtt{Rec(Y)T}} \tag{SAL-Rec}$$

$$\frac{\Gamma \vdash \mathtt{N} \ll \mathtt{Rec(X)K} \qquad \Gamma, \mathtt{Y}:*, \mathtt{X} \mathrel{<:} \mathtt{Y} \vdash \mathtt{S} \mathrel{<:} \mathtt{RBody(Y,N)}}{\Gamma \vdash \mathtt{Rec(X)S} \mathrel{<:} \mathtt{N}} \tag{SAR-Rec}$$

We shall refer to rules SA-Prod...SA-Rec as *congruence rules* and to the rules SAL-Prod...SAR-Rec as *LR-rules*. The rules named SAL-... are also called *L-rules*; the rules named SAR-... are also called *R-rules*.

5.3.2 Lemma [Weakening]: Let J be any of the algorithmic judgements introduced so far, $\mathtt{T}$ a type whose free variables are bound in Γ, and $\mathtt{X}$ a type variable not bound in Γ. If $\Gamma \vdash J$, then also $\Gamma, \mathtt{X} \mathrel{<:} \mathtt{T} \vdash J$.

Proof: Obvious induction. ∎

Notice that $\Gamma \vdash \mathtt{S} \mathrel{<:} \mathtt{T}$ does not entail that all bindings in Γ are well kinded (but it does check that $\mathtt{S}$ and $\mathtt{T}$ themselves are well kinded).

5.3.3 Lemma: If $\Gamma \vdash \mathtt{S} <: \mathtt{T}$, then $\Gamma \vdash \mathtt{S} \ll \mathtt{K}$ and $\Gamma \vdash \mathtt{T} \ll \mathtt{L}$, for some $\mathtt{K}$ and $\mathtt{L}$.

Proof: Straightforward induction. ∎

5.3.4 Definition: The *size* of a subtyping derivation is the number of subtyping rules different from SA-Promote plus the number of Eta-... rules occurring in it. Derivations of kinding premises do not affect the size.

5.3.5 Lemma [Eta and subtyping]: If $\mathcal{D} :: \Gamma \vdash \mathtt{S} =_\eta \mathtt{T}$, then $\mathcal{D}' :: \Gamma \vdash \mathtt{S} <: \mathtt{T}$ for some $\mathcal{D}'$ not larger than $\mathcal{D}$.

Proof: Easy induction on the (algorithmic) derivation $\mathcal{D}$. ∎

5.3.6 Lemma [Subtyping and parameter substitution]: If $\mathcal{D} :: \Gamma, \mathtt{X}{:}{*}, \Delta \vdash \mathtt{S} <: \mathtt{T}$ and $\Gamma \vdash \mathtt{V} \in *$ and $[\mathtt{V}/\mathtt{X}]\Delta$ is defined, then $[\mathtt{V}/\mathtt{X}]\mathtt{S}$ and $[\mathtt{V}/\mathtt{X}]\mathtt{T}$ are both defined and $\Gamma, [\mathtt{V}/\mathtt{X}]\Delta \vdash [\mathtt{V}/\mathtt{X}]\mathtt{S} <: [\mathtt{V}/\mathtt{X}]\mathtt{T}$ by a derivation not larger than $\mathcal{D}$.

Proof: We first notice that if $\mathtt{S}$ or $\mathtt{T}$ are neutral types different from $\mathtt{X}$ then so are $[\mathtt{V}/\mathtt{X}]\mathtt{S}$ and $[\mathtt{V}/\mathtt{X}]\mathtt{T}$, since a parameter cannot appear at the root of a neutral type. Next, we show by induction on a neutral type $\mathtt{N}$ that if $\mathtt{N} \in *$ then $\Gamma([\mathtt{V}/\mathtt{X}]\mathtt{N}) = [\mathtt{V}/\mathtt{X}]\Gamma(\mathtt{N})$. If, for example, $\mathtt{N} = \mathtt{EBody(U,N')}$, then either $\Gamma(\mathtt{N}')$ is neutral, in which case the induction hypothesis yields $\Gamma([\mathtt{V}/\mathtt{X}]\mathtt{N}') = [\mathtt{V}/\mathtt{X}]\Gamma(\mathtt{N}')$, so $\Gamma([\mathtt{V}/\mathtt{X}]\mathtt{N}) = \Gamma(\mathtt{EBody}([\mathtt{V}/\mathtt{X}]\mathtt{U},[\mathtt{V}/\mathtt{X}]\mathtt{N}')) = \mathtt{EBody}([\mathtt{V}/\mathtt{X}]\mathtt{U},[\mathtt{V}/\mathtt{X}]\Gamma(\mathtt{N}')) = [\mathtt{V}/\mathtt{X}]\Gamma(\mathtt{N})$. The result itself now follows by a straightforward induction on derivations. ∎

5.3.7 Lemma [Well-kindedness of promotion]: If $\Gamma \vdash \mathtt{N} \ll \mathtt{K}$, then $\Gamma \vdash \Gamma(\mathtt{N}) \ll \mathtt{K}$.

Proof: By induction on $\mathtt{N}$.

If $\mathtt{N} = \mathtt{X}$, then $\Gamma(\mathtt{X}) \ll \mathtt{K}$ by K-Var.

If $\mathtt{N} = \mathtt{N}'.\mathtt{1}$, then $\Gamma(\mathtt{N}) = \Gamma(\mathtt{N}').\mathtt{1}$ and $\mathtt{N}' \ll \mathtt{K}_1{\times}\mathtt{K}_2$. If $\Gamma(\mathtt{N}')$ is active, then it must have the form $\mathtt{T}_1{\times}\mathtt{T}_2$ and $\Gamma(\mathtt{N}) = \mathtt{T}_1$. The induction hypothesis gives $\Gamma(\mathtt{N}') \ll \mathtt{K}_1{\times}\mathtt{K}_2$, hence $\mathtt{T}_1 \ll \mathtt{K}_1$ by the definition of $\ll$, and hence the result. If $\Gamma(\mathtt{N}')$ is neutral, then $\Gamma(\mathtt{N}').\mathtt{1} \ll \mathtt{K}_1$ by K-Fst.

If $\mathtt{N} = \mathtt{EBody(U,N')}$, then $\Gamma(\mathtt{N}) = \mathtt{EBody}(\mathtt{U},\Gamma(\mathtt{N}'))$ and $\mathtt{N}' \ll \mathtt{Some(X)K'}$ and $\mathtt{U} \ll \mathtt{L}$ and $\mathtt{K} = [\mathtt{U},\mathtt{L}/\mathtt{X}]\mathtt{K}'$. The induction hypothesis gives $\Gamma(\mathtt{N}') \ll \mathtt{Some(X)K'}$. If $\Gamma(\mathtt{N}')$ is active, then $\Gamma(\mathtt{N}') = \mathtt{Some(X)T'}$ and $\mathtt{X}{:}{*} \vdash \mathtt{T}' \ll \mathtt{K}'$ by generation of $\ll$. By Lemma 5.1.1, we get $\Gamma(\mathtt{N}) = \mathtt{EBody}(\mathtt{U},\Gamma(\mathtt{N}')) = [\mathtt{U}/\mathtt{X}]\mathtt{T}' \ll [\mathtt{U},\mathtt{L}/\mathtt{X}]\mathtt{K}' = \mathtt{K}$. On the other hand, if $\Gamma(\mathtt{N}')$ is neutral, we can conclude immediately by rule K-EBody.

The other cases are similar. ∎

5.3.8 Lemma [Congruence for destructors]: Let $\mathtt{Z(Y)}$ be $\mathtt{Y.1}$, $\mathtt{Y.2}$, $\mathtt{RBody(U,Y)}$, or $\mathtt{EBody(U,Y)}$. If $\mathcal{D} :: \Gamma \vdash \mathtt{S} <: \mathtt{T}$ and $\mathtt{T} \neq \mathtt{Top}$ and $\Gamma \vdash \mathtt{Z(S)} \in *$ or $\Gamma \vdash \mathtt{Z(T)} \in *$, then $\mathcal{D}' :: \Gamma \vdash \mathtt{Z(S)} <: \mathtt{Z(T)}$ for some derivation $\mathcal{D}'$ not larger than $\mathcal{D}$.

Proof: By induction on $\mathcal{D}$. If the last rule is a congruence rule or an LR rule, then one of the immediate subderivations ends in the desired conclusion. If it is reflexivity then the conclusion is also an instance of reflexivity. The SA-Top rule has been explicitly excluded. If the last rule is an active rule then either the desired conclusion is among the premises or it can be obtained from them by invoking Lemma 5.3.6. If, for example, $\mathtt{S} = \mathtt{Some(X)S}_1$ and $\mathtt{T} = \mathtt{Some(X)T}_1$, then we must have $\mathtt{X}{:}{*} \vdash \mathtt{S}_1 <: \mathtt{T}_1$, and hence $\mathtt{EBody(U,S)} = [\mathtt{U}/\mathtt{X}]\mathtt{S}_1 <: [\mathtt{U}/\mathtt{X}]\mathtt{T}_1 = \mathtt{EBody(U,T)}$ by Lemma 5.3.6.

If the last rule is SA-Promote, then thanks to Lemma 5.3.7 we can apply the induction hypothesis to the subderivation and conclude using SA-Promote. Suppose, for example, that $\mathtt{Z(Y)} = \mathtt{RBody(U,Y)}$ and that $\mathtt{S} = \mathtt{N}$ and $\Gamma(\mathtt{N}) <: \mathtt{T}$. The induction hypothesis gives $\mathtt{RBody}(\mathtt{U},\Gamma(\mathtt{N})) <: \mathtt{RBody(U,T)}$. Since $\Gamma(\mathtt{RBody(U,N)}) = \mathtt{RBody}(\mathtt{U},\Gamma(\mathtt{N}))$, we get the desired result using SA-Promote. ∎

5.3.9 Lemma [Chain expansion]: If $\mathcal{D} :: \Gamma, \mathtt{X}{<:}\mathtt{U}, \Delta \vdash \mathtt{S} <: \mathtt{T}$, then also $\Gamma, \mathtt{Y}{<:}\mathtt{U}, \mathtt{X}{<:}\mathtt{Y}, \Delta \vdash \mathtt{S} <: \mathtt{T}$ by a derivation not larger than $\mathcal{D}$.

Proof: By induction on $\mathcal{D}$. The only interesting case is promotion of a neutral type with head variable `X`. Suppose that $\texttt{S}' = [\texttt{U}/\texttt{X}]\texttt{N}$, i.e., $\texttt{S}' = (\Gamma,\ \texttt{X<:U},\ \Delta)(\texttt{N})$ and that $\texttt{S}'$ <: `T` has been proved. The induction hypothesis gives $\texttt{S}'$ <: `T` in context Γ, `Y<:U`, `X<:Y`, Δ (by a derivation the same size or smaller), and thus [Y/X]N <: T by SA-Promote, and finally `N <: T` by another instance of SA-Promote. Conclude by recalling that instances of SA-Promote do not count toward the size of a subtyping derivation. ∎

5.3.10 Lemma [Chain contraction]: If Γ, `Y<:U`, `X<:Y`, $\Delta \vdash$ `S <: T`, then also $\mathcal{D} :: \Gamma$, `X<:U`, $[\texttt{X}/\texttt{Y}]\Delta \vdash [\texttt{X}/\texttt{Y}]\texttt{S}$ <: $[\texttt{X}/\texttt{Y}]\texttt{T}$ by a derivation not larger than $\mathcal{D}$.

Proof: By induction on $\mathcal{D}$. The only interesting case is promotion of a neutral term with head variable `X`. Suppose that $\texttt{S}' = [\texttt{Y}/\texttt{X}]\texttt{N}$ and that $\Gamma \vdash \texttt{S}'$ <: `T` has been proved. The induction hypothesis gives $[\texttt{X}/\texttt{Y}]\texttt{S}' = [\texttt{X}/\texttt{Y}]\texttt{T}$, but $[\texttt{X}/\texttt{Y}]\texttt{S}' = [\texttt{X}/\texttt{Y}]\texttt{N}$, and the conclusion follows. In other words, if $\mathcal{D}$ contains a promotion of `X` to `Y`, then this step is simply discarded in the resulting derivation. ∎

5.3.11 Theorem [Admissibility of transitivity]: If $\Gamma \vdash$ `S <: U` and $\Gamma \vdash$ `U <: T`, then $\Gamma \vdash$ `S <: T`.

Proof: By induction on the sum of the sizes of the two derivations. Let us write $\mathcal{D}_1$ and $\mathcal{D}_2$ for the derivations, LL for the last rule used in $\mathcal{D}_1$, and RR for the last rule used in $\mathcal{D}_2$. We proceed by case distinction on the form of these rules. (Note that the list of cases is not exclusive; whenever two match, use the earlier argument.)

Case: RR = S-Top

The result forms an instance of S-Top.

Case: LL = S-Top

Then RR must be S-Top too, and the result follows using S-Top.

Case: LL or RR is SA-Refl

The other derivation yields the result.

Case: Both LL and RR are instances of the same active congruence rule: SA-Prod, SA-Upd, SA-Arrow, SA-Some, SA-All, or SA-Rec

The result follows by applying the induction hypothesis to the immediate subderivations and concluding using another instance of this rule. The only slight complication arises in the case of S-Rec; we show this case explicitly.

Suppose that $\texttt{S} = \texttt{Rec(X)S}_1$ and $\texttt{U} = \texttt{Rec(X)U}_1$ and $\texttt{T} = \texttt{Rec(X)T}_1$, and that we have subderivations $\mathcal{D}_4 ::$ `Y:*`, `X<:Y` $\vdash \texttt{S}_1\texttt{(X)}$ <: $\texttt{U}_1\texttt{(Y)}$ and $\mathcal{D}_5 ::$ `Y:*`, `X<:Y` $\vdash \texttt{U}_1\texttt{(X)}$ <: $\texttt{T}_1\texttt{(Y)}$. Applying Lemma 5.3.9 to $\mathcal{D}_4$ yields `Z∈*`, `Y<:Z`, `X<:Y` $\vdash \texttt{S}_1\texttt{(X)}$ <: $\texttt{U}_1\texttt{(Y)}$, by a derivation not larger than $\mathcal{D}_4$. Applying renaming of variables and weakening to $\mathcal{D}_5$ yields a derivation (not larger than $\mathcal{D}_5$) of `Z∈*`, `Y<:Z`, `X<:Y` $\vdash \texttt{U}_1\texttt{(Y)}$ <: $\texttt{T}_1\texttt{(Z)}$. The induction hypothesis now yields `Z∈*`, `Y<:Z`, `X<:Y` $\vdash \texttt{S}_1\texttt{(X)}$ <: $\texttt{T}_1\texttt{(Z)}$, from which we obtain `Z∈*`, `X<:Z` $\vdash \texttt{S}_1\texttt{(X)}$ <: $\texttt{T}_1\texttt{(Z)}$, by identifying `Y` with `X` (Lemma 5.3.10). The conclusion follows by S-Rec.

Case: LL or RR is SA-Rec'

Then we apply Lemma 5.3.5 to the premise of the instance of SA-Rec' and use the induction hypothesis. For example, if LL is SA-Rec' and RR is SAR-Rec, then $\texttt{S} = \texttt{Rec(X)S}_1$ and $\texttt{U} = \texttt{Rec(X)U}_1$ and `T` is neutral. We have subderivations of `X:*` $\vdash \texttt{S}_1 =_\eta \texttt{U}_1$ and `Y:*`, `X<:Y` $\vdash \texttt{U}_1$ <: `RBody(Y,T)`. Lemma 5.3.5 and weakening give `Y:*`, `X<:Y` $\vdash \texttt{S}_1$ <: $\texttt{U}_1$, hence `Y:*`, `X<:Y` $\vdash \texttt{S}_1$ <: `RBody(Y,T)` by induction hypothesis and `S <: N` by SAR-Rec.

Case: LL is SA-Promote

If `S` $\ll$ `Top` then it is easy to see by inspection of the algorithmic rules that also `T` $\ll$ `Top`. Thus, the result follows by applying the induction hypothesis to the immediate subderivation of $\mathcal{D}_1$ together with $\mathcal{D}_2$, and using SA-Promote again at the end. The same strategy works if `T` is neutral (e.g., because RR is SA-Promote). Although by our convention that instances of SA-Promote do not count towards size we have no problem here because eventually rule LL will be different from SA-Promote at which point the size will get reduced. This could be formalised by adding the number of instances of SA-Promote as a low priority factor.

The remaining possibility is that $\mathtt{U}$ is neutral and $\mathtt{T}$ is not, i.e., RR is an L-rule. In this case we can apply Lemma 5.3.8 to the premise of LL, apply promotion, and then use the IH on the result and the premise of RR. Another instance of the L-rule in question then yields the result.

For a concrete example suppose that RR is SAL-Rec so $\mathtt{T} = \mathtt{Rec(X)T_1}$ and $\mathtt{U}$ is neutral and of recursive kind. The premise of RR is $\Gamma,\ \mathtt{X{:}*} \vdash \mathtt{RBody(X,U)} \mathrel{<:} \mathtt{T_1}$. Lemma 5.3.8 yields $\Gamma,\ \mathtt{X{:}*} \vdash \mathtt{RBody(X,}\Gamma\mathtt{(S))} \mathrel{<:} \mathtt{RBody(X,U)}$; SA-Promote yields $\Gamma,\ \mathtt{X{:}*} \vdash \mathtt{RBody(X,S)} \mathrel{<:} \mathtt{RBody(X,U)}$. The IH then gives $\Gamma,\ \mathtt{X{:}*} \vdash \mathtt{RBody(X,S)} \mathrel{<:} \mathtt{T_1}$, hence the result by SAL-Rec.

Case: RR is SA-Promote

Then LL must be SA-Refl, SA-Promote, or an R-rule: SAR-Prod, SAR-Upd, SAR-Some, or SAR-Rec. The first two cases have been dealt with already, so suppose that LL is an R-rule, say SAR-Some. In this case we have $\mathtt{S} = \mathtt{Some(X)S_1}$ and $\mathtt{U}$. Moreover, we have the following subderivation:

$\mathcal{D}_3 :: \mathtt{X{:}*} \vdash \mathtt{S_1} \mathrel{<:} \mathtt{EBody(X,U)}$.

Now, if $\mathtt{T}$ is $\mathtt{Top}$ then the desired conclusion can be obtained using SA-Top. Otherwise, we may apply Lemma 5.3.8 to $\mathcal{D}_2$, yielding $\mathtt{X{:}*} \vdash \mathtt{EBody(X,U)} \mathrel{<:} \mathtt{EBody(X,T)}$. The induction hypothesis gives us $\mathtt{X{:}*} \vdash \mathtt{S_1} \mathrel{<:} \mathtt{EBody(X,T)}$. We conclude by S-Some or SAL-Some, according to whether $\mathtt{T}$ is active or not.

Case: LL is an R-rule

Then the only remaining possibility is that RR is an L-rule for the same type former as LL. The most difficult case arises when this type former is $\mathtt{Rec}$, so we use this as an illustrative example.

Suppose we have $\mathtt{S} = \mathtt{Rec(X)S_1}$, $\mathtt{U}$ neutral, and $\mathtt{T} = \mathtt{Rec(X)T_1}$. We have subderivations

$\mathcal{D}_4 :: \mathtt{Y{:}*},\ \mathtt{X{<:}Y} \vdash \mathtt{S_1(X)} \mathrel{<:} \mathtt{RBody(Y,U)}$
$\mathcal{D}_5 :: \mathtt{Y{:}*},\ \mathtt{X{<:}Y} \vdash \mathtt{RBody(X,U)} \mathrel{<:} \mathtt{T_1(Y)}$
$\mathcal{D}_6 :: \mathtt{U} \mathrel{<:} \mathtt{Rec(X)P}$.

We now proceed as in the S-Rec case, this time using $\mathtt{RBody(Y,U)}$ as cut-formula.

Case: LL is an L-rule

Then RR is either an R-rule for the same former or the corresponding congruence rule. In each case we can apply the induction hypothesis to the subderivations and proceed as in the previous case. ∎

Our job for the remainder of this section is to prove a substitution lemma for the algorithmic subtyping relation (Proposition 5.3.20). For this purpose, we introduce an auxiliary *refinement relation* on kinds—something like the subtyping relation but with a pointwise clause for recursive types to match their kinding rule.

5.3.12 Definition [Kind refinement]: The relation $\lll:$ between kinds is defined as follows:

$$\frac{}{\Gamma \vdash \mathtt{K} \lll: \mathtt{Top}} \qquad \text{(Ref-Top)}$$

$$\frac{}{\Gamma \vdash \mathtt{X} \lll: \mathtt{X}} \qquad \text{(Ref-Refl)}$$

$$\frac{\Gamma \vdash \mathtt{K_1} \lll: \mathtt{L_1} \qquad \Gamma \vdash \mathtt{K_2} \lll: \mathtt{L_2}}{\Gamma \vdash \mathtt{K_1}{\times}\mathtt{K_2} \lll: \mathtt{L_1}{\times}\mathtt{L_2}} \qquad \text{(Ref-Prod)}$$

$$\frac{\Gamma \vdash \mathtt{S} =_\eta \mathtt{T} \qquad \Gamma \vdash \mathtt{K} \lll: \mathtt{L}}{\Gamma \vdash \mathtt{!S}{\times}\mathtt{K} \lll: \mathtt{!T}{\times}\mathtt{L}} \qquad \text{(Ref-Upd)}$$

$$\frac{\Gamma,\ \mathtt{X{:}*} \vdash \mathtt{K} \lll: \mathtt{L}}{\Gamma \vdash \mathtt{Some(X)K} \lll: \mathtt{Some(X)L}} \qquad \text{(Ref-Some)}$$

$$\frac{\Gamma,\ \mathtt{X{:}*} \vdash \mathtt{K} \lll: \mathtt{L}}{\Gamma \vdash \mathtt{Rec(X)K} \lll: \mathtt{Rec(X)L}} \qquad \text{(Ref-Rec)}$$

5.3.13 Lemma [Transitivity of refinement]: Kind refinement is transitive.

Proof: Easy induction on derivations. ∎

5.3.14 Lemma [Reflexivity of refinement]: If $\Gamma \vdash \mathtt{K} \ll \mathtt{K}$, then $\Gamma \vdash \mathtt{K} \lll: \mathtt{K}$.

Proof: Easy induction on derivations. ∎

5.3.15 Lemma [Monotonicity of refinement]: If $\mathtt{X}{:}* \vdash \mathtt{K}_1 \lll: \mathtt{L}_1$, and if $\mathtt{T}$ is any type and $\mathtt{K}_2 \lll: \mathtt{L}_2$, then $[\mathtt{T},\mathtt{K}_2/\mathtt{X}]\mathtt{K}_1 \lll: [\mathtt{T},\mathtt{L}_2/\mathtt{X}]\mathtt{L}_1$.

Proof: By induction on the structure of $\mathtt{L}_1$. If $\mathtt{L}_1 = \mathtt{Top}$, then the result follows by REF-TOP. If $\mathtt{L}_1 = \mathtt{X}$, then $\mathtt{K}_1$ must be $\mathtt{X}$ too, and the result follows from the assumption. If $\mathtt{L}_1 = \mathtt{!S}_1{\times}\mathtt{L1}'$, then $\mathtt{K}_1 = \mathtt{!T}_1{\times}\mathtt{K}_1{}'$ and $\mathtt{S}_1 =_\eta \mathtt{T}_1$ and $\mathtt{K}_1{}' \lll: \mathtt{L}_1{}'$. Proposition 5.2.2 and the induction hypothesis together with REF-UPD then yield the result. The other cases are similar. ∎

5.3.16 Lemma [Kinding and subtyping]: If $\Gamma \vdash \mathtt{S} <: \mathtt{T}$ and $\Gamma \vdash \mathtt{S} \ll \mathtt{K}$ and $\Gamma \vdash \mathtt{T} \ll \mathtt{L}$ then $\Gamma \vdash \mathtt{K} \lll: \mathtt{L}$.

Proof: By induction on a derivation of $\Gamma \vdash \mathtt{S} <: \mathtt{T}$.

If the derivation is an instance of SA-REFL, Lemma 5.3.14 yields the result.

If the derivation is an instance of SA-TOP, use REF-TOP.

In the case of rules SA-PROD to SA-REC', the result follows by applying the induction hypothesis to the premises. The most difficult of these cases is SA-REC. Suppose, therefore, that $\mathtt{S} = \mathtt{Rec(X)S}_1$ and $\mathtt{T} = \mathtt{Rec(Y)T}_1$ and $\mathtt{K} = \mathtt{Rec(X)K}_1$ and $\mathtt{L} = \mathtt{Rec(Y)L}_1$ and $\mathtt{X}{:}* \vdash \mathtt{S}_1 \ll \mathtt{K}_1$ and $\mathtt{Y}{:}* \vdash \mathtt{T}_1 \ll \mathtt{L}_1$ and, finally, $\mathtt{Y}{:}*,\ \mathtt{X}{<:}\mathtt{Y} \vdash \mathtt{S}_1 <: \mathtt{T}_1$. Now Lemma 5.3.10 yields a derivation of $\mathtt{Y}{:}* \vdash \mathtt{S}_1 <: \mathtt{T}_1$. The induction hypothesis gives $\mathtt{Y}{:}* \vdash \mathtt{K}_1 \lll: \mathtt{L}_1$, hence $\mathtt{K} \lll: \mathtt{L}$ by REF-REC.

On the other hand, suppose that $\mathtt{S} <: \mathtt{T}$ was derived by rule SAL-SOME. Then $\mathtt{S}$ is neutral and $\mathtt{T} = \mathtt{Some(X)T}_1$ and $\mathtt{L} = \mathtt{Some(X)L}_1$ and $\mathtt{X}{:}* \vdash \mathtt{T}_1 \ll \mathtt{L}_1$ and $\mathtt{K} = \mathtt{Some(X)K}_1$ and $\mathtt{X}{:}* \vdash \mathtt{EBody(X,S)} <: \mathtt{T}$. Now rule K-EBODY gives $\mathtt{X}{:}* \vdash \mathtt{EBody(X,S)} \ll [\mathtt{X},\mathtt{X}/\mathtt{X}]\mathtt{K}_1 = \mathtt{K}_1$. Therefore $\mathtt{X}{:}* \vdash \mathtt{K}_1 <: \mathtt{L}_1$ by the induction hypothesis, and $\mathtt{K} \lll: \mathtt{L}$ by REF-SOME. The other LR-rules are similar.

Finally, in the case of SA-PROMOTE, we invoke Lemma 5.3.7 and the induction hypothesis. ∎

5.3.17 Lemma [Kinding and substitution, general case]: Suppose that $\Gamma,\ \mathtt{X}{<:}\mathtt{U},\ \Delta \vdash \mathtt{S} \ll \mathtt{K}$, that $\Gamma \vdash \mathtt{V} <: \mathtt{U}$, and that $[\mathtt{V}/\mathtt{X}]\Delta$ is defined. Then $[\mathtt{V}/\mathtt{X}]\mathtt{S}$ is defined and $\Gamma,\ [\mathtt{V}/\mathtt{X}]\Delta \vdash [\mathtt{V}/\mathtt{X}]\mathtt{S} \ll \mathtt{L}$ for some $\mathtt{L} \lll: [\mathtt{V}/\mathtt{X}]\mathtt{K}$. (Notice that since $\mathtt{X}$ is not a parameter in can only occur in the invariant position of a updatable product in $\mathtt{K}$).

Proof: By induction on the derivation of $\mathtt{S} \ll \mathtt{K}$.

If $\mathtt{S}$ is active, then the result follows by applying the induction hypothesis to the premises. Consider, for example, the case $\mathtt{S} = \mathtt{Rec(Y)S}_1$ and $\mathtt{K} = \mathtt{Rec(Y)K}_1$ and $\mathtt{Y}{:}* \vdash \mathtt{S}_1 \ll \mathtt{K}_1$. The induction hypothesis gives $\mathtt{L}_1$ such that $\mathtt{Y}{:}* \vdash [\mathtt{V}/\mathtt{X}]\mathtt{S}_1 \ll \mathtt{L}_1 \lll: [\mathtt{V}/\mathtt{X}]\mathtt{K}_1$. Hence $[\mathtt{V}/\mathtt{X}]\mathtt{S} \ll \mathtt{Rec(Y)L}_1 \lll: [\mathtt{V}/\mathtt{X}]\mathtt{K}$ by K-REC and REF-REC.

If $\mathtt{S} = \mathtt{X}$, then the result follows from Lemmas 5.3.3 and 5.3.16. If $\mathtt{S} = \mathtt{Y} \neq \mathtt{X}$, then the result follows by applying the induction hypothesis to the bound of $\mathtt{Y}$.

Finally, consider the case where $\mathtt{S} = \mathtt{EBody(T,N)}$, as an example of the LR-rules. Then $\mathtt{T} \ll \mathtt{K}_1$ and $\mathtt{N} \ll \mathtt{Some(Y)K}_2$ and $\mathtt{K} = [\mathtt{T},\mathtt{K}_1/\mathtt{Y}]\mathtt{K}_2$. The induction hypothesis yields $[\mathtt{V}/\mathtt{X}]\mathtt{T} \ll \mathtt{L}_1 \lll: [\mathtt{V}/\mathtt{X}]\mathtt{K}_1$ and $[\mathtt{V}/\mathtt{X}]\mathtt{N} \ll \mathtt{Some(Y)L}_2 \lll: \mathtt{Some(Y)}[\mathtt{V}/\mathtt{X}]\mathtt{K}_2$ for some $\mathtt{L}_1$ and $\mathtt{L}_2$. Now we have two cases to distinguish. Either $[\mathtt{V}/\mathtt{X}]\mathtt{N}$ is still neutral, in which case $[\mathtt{V}/\mathtt{X}]\mathtt{S} \ll [[\mathtt{V}/\mathtt{X}]\mathtt{T},\mathtt{L}_1/\mathtt{Y}]\mathtt{L}_2$, or else $[\mathtt{V}/\mathtt{X}]\mathtt{N} = \mathtt{Some(Y)P}$, where $\mathtt{Y}{:}* \vdash \mathtt{P} \ll \mathtt{L}_2$. In this case, $[\mathtt{V}/\mathtt{X}]\mathtt{S} = [[\mathtt{V}/\mathtt{X}]\mathtt{T}/\mathtt{Y}]\mathtt{P} \ll [[\mathtt{V}/\mathtt{X}]\mathtt{T},\mathtt{L}_1/\mathtt{Y}]\mathtt{L}_2$ by Lemma 5.1.1. So in either case $[\mathtt{V}/\mathtt{X}]\mathtt{S}$ has kind $\mathtt{L} \stackrel{\text{def}}{=} [[\mathtt{V}/\mathtt{X}]\mathtt{T},\mathtt{L}_1/\mathtt{Y}]\mathtt{L}_2$. But $\mathtt{L} \lll: [\mathtt{V}/\mathtt{X}]\mathtt{K}$ by Lemma 5.3.15, hence the result. ∎

5.3.18 Proposition [Eta and substitution, general case]: If $\mathcal{D} :: \Gamma,\ \mathtt{X}{<:}\mathtt{U},\ \Delta \vdash^a \mathtt{S} =_\eta \mathtt{T}$ and $\Gamma \vdash \mathtt{V} <: \mathtt{U}$ and $[\mathtt{V}/\mathtt{X}]\Delta$ is defined, then $\Gamma,\ [\mathtt{V}/\mathtt{X}]\Delta \vdash^a [\mathtt{V}/\mathtt{X}]\mathtt{S} =_\eta [\mathtt{V}/\mathtt{X}]\mathtt{T}$.

Proof: Induction on derivations. The congruence rules are straightforward applications of the induction hypothesis. For ETA-A-REFL we use Lemma 5.3.17. For the LR-rules we proceed as usual by case distinction on whether the substituted types are still neutral or not. In the case of ETA-AL(R)-SOME and ETA-AL(R)-REC we use Prop. 5.2.2. Let us look at rule ETA-AL-REC. In this case `S` is neutral of recursive kind and `T` is $\texttt{Rec(Y)T}_1$. We also know that Γ, , `X<:U`, `Y:*` $\vdash^a$ `RBody(Y,S) <:` $\texttt{T}_1$. The induction hypothesis gives Γ, `Y:*` $\vdash^a$ `RBody(Y,[V/X]S)` $=_\eta$ $\texttt{[V/X]T}_1$. Lemma 5.3.17 together with the definition of kind refinement shows that `[V/X]S` is still of recursive kind. If `[V/X]S` is neutral then the desired result follows using ETA-AL-REC. Otherwise, $\texttt{[V/X]S=Rec(Y)S}_1$ for some type $\texttt{S}_1$ and $\texttt{RBody(Y,[V/X]S)=S}_1$. The result follows with ETA-A-REC. ∎

5.3.19 Lemma [Substitutivity of promotion]: Let $\Sigma = \Gamma$, `X<:U`, Δ. If $\Sigma \vdash$ `N` $\in *$ and $\Gamma \vdash$ `V <: U` and $\texttt{[V/X]}\Delta$ is defined, then Γ, $\texttt{[V/X]}\Delta \vdash$ `[V/X]N <:` $\texttt{[V/X]}\Sigma\texttt{(N)}$.

Proof: First notice that, by the form of the definition of promotion, $\Sigma \vdash \texttt{N} \in *$ implies $\Sigma \vdash \Sigma(\texttt{N}) \in *$; from these two facts, Lemma 5.3.17 tells us that `[V/X]N` and $\texttt{[V/X]}\Gamma\texttt{(N)}$ are both defined and well-kinded. Now proceed by induction on the form of N. If `N` = `X`, then `[V/X]N` = `V` and $\texttt{[V/X]}\Sigma\texttt{(N)}$ = `[V/X]U` = `U`, since `X` is not free in `U`. Hence, the result follows by the assumption on `V`. If `N` = `Y`, then the result follows using SA-PROMOTE on `N` and reflexivity (on $\texttt{[V/X]}\Sigma\texttt{(Y)}$). In all other cases the result follows by applying Lemma 5.3.8 to the induction hypothesis. ∎

5.3.20 Proposition [Substitutivity of subtyping]: If Γ, `X<:U`, $\Delta \vdash$ `S <: T` and $\Gamma \vdash$ `V <: U` and $\texttt{[V/X]}\Delta$ is defined, then Γ, $\texttt{[V/X]}\Delta \vdash$ `[V/X]S <: [V/X]T`.

Proof: By induction on a derivation of Γ, `X<:U`, $\Delta \vdash$ `S <: T`.

If the last rule is SA-REFL or SA-TOP, then the result follows using the same rule (plus Lemma 5.3.17 to establish the required kinding premise).

The congruence rules for active types commute with substitution directly, so the result follows by applying the induction hypothesis to the subderivations and using the same rule on the results. The arguments for SA-UPD and SA-REC' use Proposition 5.3.18. SA-ALL uses Lemma 5.3.17 for the kinding premise.

If the last rule is an LR-rule, then we use Lemma 5.3.17 on the kinding premises and a case distinction on whether the substituted type is still neutral or not like in the proof of Prop. 5.3.18. Suppose, for example, that the rule is SAL-REC; then `S` is neutral and `T` = $\texttt{Rec(Y)T}_1$ and `S` $\ll$ `Rec(Y)K` and `Y:*`, `Z<:Y` $\vdash$ `RBody(Z,S) <:` $\texttt{T}_1$. If `[V/X]S` is still neutral, then Lemma 5.3.17 together with the definition of kind refinement shows that `[V/X]S` has recursive kind so the result follows by applying SAL-REC to the induction hypothesis. Otherwise, `[V/X]S` equals $\texttt{Rec(Z)S}_1$ for some type $\texttt{S}_1$ and `RBody(Z,[V/X]S)` = $\texttt{S}_1$ by definition of substitution. The result then follows from rule SA-REC.

Finally, if `S <: T` has been derived by SA-PROMOTE, i.e., `S` is neutral and $\Gamma\texttt{(S)}$ `<: T`, then we obtain `[V/X]S <:` $\texttt{[V/X]}\Gamma\texttt{(S)}$ from Lemma 5.3.19. Theorem 5.3.11 and the induction hypothesis applied to the immediate subderivation then yield the result. ∎

5.4 Soundness and Completeness of Algorithmic Subtyping

We now write $\vdash^a$ for algorithmic derivations and $\vdash$ for derivations in the declarative systems.

5.4.1 Theorem [Soundness of algorithmic subtyping]:

1. If $\Gamma \vdash$ `N` $\in *$, then $\Gamma \vdash$ `N <:` $\Gamma\texttt{(N)}$.
2. If $\Gamma \vdash^a$ `S <: T`, then $\Gamma \vdash$ `S <: T`.

Proof: Straightforward induction. Use S-TRANS for soundness of SA-PROMOTE. ∎

5.4.2 Theorem [Completeness of algorithmic subtyping]: If $\Gamma \vdash$ `S <: T`, then $\Gamma \vdash^a$ `S <: T`

Proof: Putting together the lemmas and propositions from above. ∎

Before we consider decidability, let us pause to discharge a pending proof obligation from Section 3.3.

Proof of Theorem 3.3.1: Suppose that $\Gamma \vdash \mathtt{S} <: \mathtt{!T_1{\times}T_2}$. By Theorem 5.4.2, $\Gamma \vdash^a \mathtt{S} <: \mathtt{!T_1{\times}T_2}$. Clearly, $\mathtt{!T_1{\times}T_2}$ must be well kinded, but by generation of kinding the only possible kind is $\mathtt{T_1{\times}K_1}$. Lemma 5.3.16 then entails that $\mathtt{S}$ has some kind $\mathtt{K} \lll: \mathtt{!T_1{\times}K_1}$. Generation of refinement then yields $\mathtt{K} = \mathtt{!T_1'{\times}K'}$ where $\mathtt{T_1}' =_\eta \mathtt{T_1}$, hence the result. The other cases are analogous. ∎

5.5 Decidability

We have already established soundness and completeness results for our algorithmic presentations of kinding, eta-conversion, and subtyping. To show that these relations are decidable, it only remains to show that the algorithms terminate on all inputs. (The algorithmic typing relation defined in Section 5.6 will also have this property, by an easy inspection.)

5.5.1 Proposition [Kind checking is decidable]: The algorithm resulting from inverting the kinding rules terminates on all inputs.

Proof: Let Γ be a context and $\mathtt{T}$ a type. We define $w(\Gamma, \mathtt{T})$ as the length of $\mathtt{T}$ plus the length of the part of Γ which binds free variables in $\mathtt{T}$. An inspection of the kinding rules then shows that the measure w of the conclusion of a kinding rule is strictly larger than the measure of any of its premises. Thus termination follows by induction on w. ∎

5.5.2 Proposition [Eta-equality is decidable]: The algorithm resulting from inverting the algorithmic eta-rules terminates on all inputs.

Proof: We assign to an instance $\Gamma \vdash \mathtt{S} =_\eta \mathtt{T}$ the number of active type formers contained in $\mathtt{S}$ and $\mathtt{T}$. This measure is reduced by every backwards application of a rule. ∎

To show that the subtyping algorithm terminates, we need to do a little more work. We define a translation of $F_{\leq}^{TD}$ types into terms of a simply-typed lambda calculus with product types, function types, and Top. Then we reduce termination of the subtyping algorithm to strong normalisation of this lambda calculus, which is well-known.

Let λ_M be the fragment of $F_{\leq}^{TD}$ generated by the type formers Top, $\to$, and $\times$ (no type destructors). It follows by standard methods that this calculus is strongly normalising, i.e., that there does not exist an infinite reduction sequence starting from a well-typed term in λ_M. For example, the normalisation proof for Gödel's system T given in [GLT89] readily extends to λ_M by interpreting subtyping as inclusion of reducibility sets and interpreting Top as the set of strongly normalising terms.

The translation $(-)^*$ from $F_{\leq}^{TD}$-kinds to λ_M-types is defined by:

$$
\begin{array}{lcl}
\mathtt{Top}^* & = & \mathtt{Top} \\
\mathtt{X}^* & = & \mathtt{Top} \\
\mathtt{K_1{\times}K_2}^* & = & \mathtt{K_1}^*{\times}\mathtt{K_2}^* \\
\mathtt{!K_1{\times}K_2}^* & = & \mathtt{K_1}^*{\times}\mathtt{K_2}^* \\
\mathtt{Some(X)K}^* & = & \mathtt{Top}{\to}\mathtt{K}^* \\
\mathtt{Rec(X)K}^* & = & \mathtt{Top}{\to}\mathtt{K}^*
\end{array}
$$

Let Γ be a $F_{\leq}^{TD}$-context. We translate a type $\mathtt{T}$ with $\Gamma \vdash \mathtt{T} \ll \mathtt{K}$ to an λ_M-term of type $\mathtt{K}^*$ having the *parameters* of Γ as free variables of type Top and no other free variables. The defining clauses are as follows, where I stands for an appropriately typed identity function `fun(x:T)x`.

$$
\begin{array}{lcll}
(\mathtt{X})^*_\Gamma & = & \mathtt{X} & \text{if } \Gamma(\mathtt{X}) = * \\
(\mathtt{X})^*_\Gamma & = & \mathtt{I}\ (\Gamma(\mathtt{X}))^*_\Gamma & \text{otherwise} \\
(\mathtt{T_1}\Diamond\mathtt{T_2})^*_\Gamma & = & \mathtt{I}\ (\mathtt{I}\ ((\mathtt{T_1})^*_\Gamma, (\mathtt{T_2})^*_\Gamma)) & \text{where } \Diamond \in \{\to, \times, !\ldots\times\} \\
(\mathtt{All(X{<:}T_1)T_2})^*_\Gamma & = & \mathtt{I}\ (\mathtt{T_2})^*_{\Gamma,\mathtt{X<:T_1}} & \\
(\mathtt{Some(X)T})^*_\Gamma & = & \mathtt{I}\ (\mathtt{I}\ (\mathtt{fun(X:Top)}(\mathtt{T})^*_{\Gamma,\mathtt{X:*}})) & \\
(\mathtt{Rec(X)T})^*_\Gamma & = & \mathtt{I}\ (\mathtt{I}\ (\mathtt{fun(Y:Top)}\ (\mathtt{fun(X:Top)}(\mathtt{T})^*_{\Gamma,\mathtt{X:*}})\ (\mathtt{I\ Y}))) & \\
(\mathtt{N.1})^*_\Gamma & = & (\mathtt{N})^*_\Gamma\mathtt{.1} & \\
(\mathtt{N.2})^*_\Gamma & = & (\mathtt{N})^*_\Gamma\mathtt{.2} & \\
(\mathtt{EBody(T,N)})^*_\Gamma & = & (\mathtt{N})^*_\Gamma(\mathtt{T})^*_\Gamma & \\
(\mathtt{RBody(T,N)})^*_\Gamma & = & (\mathtt{N})^*_\Gamma(\mathtt{T})^*_\Gamma &
\end{array}
$$

The compositional definition of this translation immediately yields the following substitution property:

5.5.3 Lemma: If $\Gamma, \mathtt{X}{:}{*}, \Delta \vdash \mathtt{T} \in *$ and $\Gamma \vdash \mathtt{U} \in *$, then

$$[(\mathtt{U})^*_\Gamma \,/\, \mathtt{X}](\mathtt{T})^*_{\Gamma,\mathtt{X}:*,\Delta} = ([\mathtt{U}/\mathtt{X}]\mathtt{T})^*_{\Gamma,[\mathtt{U}/\mathtt{X}]\Delta}.$$

If $\mathtt{e}$ is an λ_M-term, write $\mu(\mathtt{e})$ for the length of the longest reduction sequence starting from $\mathtt{e}$ and leading to a normal form.

5.5.4 Lemma: If $\Gamma \vdash \mathtt{N} \in *$ and $\mathtt{N}$ is not a parameter, then $\mathtt{N}$ reduces in a nonzero number of steps to $\Gamma(\mathtt{N})$, hence $\mu(\Gamma(\mathtt{N}))^*_\Gamma < \mu(\mathtt{N})^*_\Gamma$.

Proof: If $\mathtt{N}$ is a variable then this is immediate from the definition. In all other cases, the result follows directly from the induction hypothesis and Lemma 5.5.3.

Consider, for example, the case $\mathtt{N} = \mathtt{RBody(U,N_0)}$. If $\Gamma(\mathtt{N_0})$ is neutral, then $(\Gamma(\mathtt{N}))^*_\Gamma = (\Gamma(\mathtt{N_0}))^*_\Gamma(\mathtt{U})^*_\Gamma$. So since, by the induction hypothesis, $\mathtt{N_0}$ reduces to $\Gamma(\mathtt{N_0})$, we get the desired result by applying the same reduction to the head of the application.

If, on the other hand, $\Gamma(\mathtt{N_0})$ equals $\mathtt{Rec(X)T}$, then $\Gamma(\mathtt{N})$ is $[\mathtt{U}/\mathtt{X}]\mathtt{T}$ and, by Lemma 5.5.3, $(\Gamma(\mathtt{N}))^*_\Gamma$ equals $[(\mathtt{U})^*_\Gamma \,/\, \mathtt{X}](\mathtt{T})^*_\Gamma$. Now $\mathtt{N}$ can be reduced to this term using the reduction sequence from the induction hypothesis followed by five reductions. ∎

5.5.5 Lemma: Suppose that $\Gamma_0 \vdash \mathtt{S_0} <: \mathtt{T_0}$ appears as immediate premise of $\Gamma \vdash \mathtt{S} <: \mathtt{T}$ in one of the algorithmic subtyping rules. Then

$$\mu(\mathtt{S_0})^*_{\Gamma_0} + \mu(\mathtt{T_0})^*_{\Gamma_0} < \mu(\mathtt{S})^*_\Gamma + \mu(\mathtt{T})^*_\Gamma$$

Proof: For SA-PROMOTE we use Lemma 5.5.4. The congruence rules require straightforward calculations from the definitions and Lemma 5.5.3. Let us look at the most complex one: SA-REC. Let $\mathtt{e_0}, \mathtt{e_0}'$ be the translations of $\mathtt{S_0}, \mathtt{T_0}$ and $\mathtt{e}, \mathtt{e}'$ the translations of $\mathtt{S}, \mathtt{T}$. Furthermore, let $\mathtt{e_1}$ be $(\mathtt{S_0})^*_{\Gamma,\mathtt{X}:*}$ and $\mathtt{e}_1'$ be $(\mathtt{T_0})^*_{\Gamma,\mathtt{Y}:*}$. We have

$$\begin{array}{lcl}
\mathtt{e} & = & \mathtt{I(I(fun(Y:Top)(fun(X:Top)e_1)(I\ Y)))} \\
\mathtt{e}' & = & \mathtt{I(I(fun(Y:Top)(fun(X:Top)e_1{}')(I\ Y)))} \\
\mathtt{e_0} & = & [\mathtt{I\ X/Y}]\mathtt{e_1} \\
\mathtt{e_0}' & = & \mathtt{e_1}'.
\end{array}$$

The third and fourth equation follow by inspection of the treatment of variables in the translation.

This analysis shows that $\mu(\mathtt{e}) \geq 3 + \mu(\mathtt{e_1})$ and $\mu(\mathtt{e}') \geq 3 + \mu(\mathtt{e_1}')$ and $\mu(\mathtt{e_0}) \leq \mu(\mathtt{e_1}) + 1$, hence $\mu(\mathtt{e}) + \mu(\mathtt{e}') \geq 6 + \mu(\mathtt{e_1}) + \mu(\mathtt{e_1}') \geq 5 + \mu(\mathtt{e_0}) + \mu(\mathtt{e_0}') > \mu(\mathtt{e_0}) + \mu(\mathtt{e_0}')$.

The most interesting cases are the LR-rules. We show the most difficult one: SAL-REC. Let $\mathtt{e_0}, \mathtt{e_0}'$ be the translations of $\mathtt{S_0}, \mathtt{T_0}$ and $\mathtt{e}, \mathtt{e}'$ be the translations of $\mathtt{S}, \mathtt{T}$. We have

$$\begin{array}{lcl}
\mathtt{e}' & = & \mathtt{I(I(fun(Y:Top)(fun(X:Top)e_0{}')(I\ Y)))} \\
\mathtt{e_0} & = & \mathtt{e\ (I\ Y)}.
\end{array}$$

This shows that $\mu(\mathtt{e}') \geq \mu(\mathtt{e_0}') + 3$ and $\mu(\mathtt{e_0}) \leq \mu(\mathtt{e}) + 2$, hence the result. ∎

As an immediate corollary we now obtain the desired result.

5.5.6 Theorem: Subtyping is decidable.

Proof: In order to decide whether $\Gamma \vdash \mathtt{S} <: \mathtt{T}$, apply the algorithmic subtyping rules backwards until either a proof is found or no rule applies anymore. This process terminates due to strong normalisation of λ_M and Lemma 5.5.5. ∎

We note, in passing, that this proof gives us a very bad upper bound on the complexity of the subtyping procedure (elementary or worse). Observe, however, that all abstractions occurring in translations of $F^{TD}_{\leq}$-types are of type Top so that a variable never appears in applied position. We believe that normalisation for this fragment of λ_M is of more reasonable complexity (exponential or better), but we haven't looked into details.

5.6 Algorithmic Typing

In order to decide typechecking we introduce a set of syntax-directed typing rules which (when read from bottom to top, as a "logic program") compute the minimal type of a given term in a given context.

5.6.1 Definition: The judgement $\Gamma \vdash \mathtt{T} \uparrow \mathtt{A}$ read "the least active supertype of $\mathtt{T} \ll \mathtt{Top}$ is $\mathtt{A}$" is defined by

$$\frac{\Gamma \vdash \mathtt{A} \ll \mathtt{Top}}{\Gamma \vdash \mathtt{A} \uparrow \mathtt{A}}$$

$$\frac{\Gamma \vdash \Gamma(\mathtt{N}) \uparrow \mathtt{A}}{\Gamma \vdash \mathtt{N} \uparrow \mathtt{A}}$$

.

5.6.2 Proposition: If $\Gamma \vdash \mathtt{T} \uparrow \mathtt{A}$ then $\Gamma \vdash \mathtt{T} \mathrel{\mathtt{<:}} \mathtt{A}$. If, moreover $\Gamma \vdash \mathtt{T} \mathrel{\mathtt{<:}} \mathtt{A}'$ then $\Gamma \vdash \mathtt{A} \mathrel{\mathtt{<:}} \mathtt{A}'$.

Proof: The first part is an easy induction on the definition of $\Gamma \vdash \mathtt{T} \uparrow \mathtt{A}$; the second an easy induction on an algorithmic subtyping derivation of $\Gamma \vdash \mathtt{T} \mathrel{\mathtt{<:}} \mathtt{A}'$. ∎

Note that, if $\Gamma \vdash \mathtt{N} \uparrow \mathtt{A}$, then $\mathtt{A}$ is $\mathtt{Int}$ or $\mathtt{Top}$ or of the form $\mathtt{T}_1 \to \mathtt{T}_2$ or $\mathtt{All(X<:T_1)T_2}$.

The algorithmic typing rules are now as follows.

$$\frac{\Gamma \text{ well formed}}{\Gamma \vdash \mathtt{x} \in \Gamma(\mathtt{x})} \qquad \text{(TA-Var)}$$

$$\frac{\Gamma \text{ well formed}}{\Gamma \vdash \mathtt{i} \in \mathtt{Int}} \qquad \text{(TA-Const)}$$

$$\frac{\Gamma, \mathtt{x}:\mathtt{T}_1 \vdash \mathtt{e} \in \mathtt{T}_2}{\Gamma \vdash \mathtt{fun(x:T_1)e} \in \mathtt{T}_1 \to \mathtt{T}_2} \qquad \text{(TA-Abs)}$$

$$\frac{\Gamma \vdash \mathtt{e}_1 \in \mathtt{T}_1 \qquad \Gamma \vdash \mathtt{e}_2 \in \mathtt{T}_2 \qquad \Gamma \vdash \mathtt{T}_1 \uparrow \mathtt{U} \to \mathtt{T} \qquad \Gamma \vdash \mathtt{T}_2 \mathrel{\mathtt{<:}} \mathtt{U}}{\Gamma \vdash (\mathtt{e}_1\ \mathtt{e}_2) \in \mathtt{T}} \qquad \text{(TA-App)}$$

$$\frac{\Gamma, \mathtt{X<:T_1} \vdash \mathtt{e} \in \mathtt{T}_2}{\Gamma \vdash \mathtt{fun(X<:T_1)e} \in \mathtt{All(X<:T_1)T_2}} \qquad \text{(TA-TAbs)}$$

$$\frac{\Gamma \vdash \mathtt{e} \in \mathtt{T} \qquad \Gamma \vdash \mathtt{T} \uparrow \mathtt{All(X<:T_1)T_2} \qquad \Gamma \vdash \mathtt{U} \mathrel{\mathtt{<:}} \mathtt{T}_1}{\Gamma \vdash \mathtt{e[U]} \in \mathtt{[U/X]T_2}} \qquad \text{(TA-TApp)}$$

$$\frac{\Gamma \vdash \mathtt{e}_1 \in \mathtt{T}_1 \qquad \mathtt{e}_2 \in \mathtt{T}_2}{\Gamma \vdash (\mathtt{e}_1,\mathtt{e}_2) \in \mathtt{T}_1 \times \mathtt{T}_2} \qquad \text{(TA-Pair)}$$

$$\frac{\Gamma \vdash \mathtt{e}_1 \in \mathtt{T}_1 \qquad \mathtt{e}_2 \in \mathtt{T}_2}{\Gamma \vdash (!\mathtt{e}_1,\mathtt{e}_2) \in !\mathtt{T}_1 \times \mathtt{T}_2} \qquad \text{(TA-Pair-Upd)}$$

$$\frac{\Gamma \vdash \mathtt{e} \in \mathtt{T} \qquad \mathtt{T} \ll \mathtt{K}_1 \times \mathtt{K}_2 \qquad \mathtt{i=1,2}}{\Gamma \vdash \mathtt{e.i} \in \mathtt{T.i}} \qquad \text{(TA-Proj)}$$

$$\frac{\Gamma \vdash \mathtt{e} \in \mathtt{T} \qquad \mathtt{T} \ll !\mathtt{T}_1 \times \mathtt{K}_2}{\Gamma \vdash \mathtt{e.1} \in \mathtt{T}_1} \qquad \text{(TA-Proj-Upd-1)}$$

$$\frac{\Gamma \vdash \mathtt{e} \in \mathtt{T} \qquad \mathtt{T} \ll \mathtt{!T_1{\times}K_2}}{\Gamma \vdash \mathtt{e.2} \in \mathtt{T.2}} \qquad \text{(TA-Proj-Upd-2)}$$

$$\frac{\Gamma \vdash \mathtt{T} \ll \mathtt{Some(X)K} \qquad \Gamma \vdash \mathtt{S} \in * \qquad \Gamma \vdash \mathtt{e} \in \mathtt{EBody(S,T)}}{\Gamma \vdash \mathtt{pack\ [S,e]\ as\ T} \in \mathtt{T}} \qquad \text{(TA-Pack)}$$

$$\frac{\Gamma \vdash \mathtt{e} \in \mathtt{T} \qquad \Gamma \vdash \mathtt{T} \ll \mathtt{Some(X)K} \qquad \Gamma,\mathtt{X{:}*},\mathtt{x{:}EBody(X,T)} \vdash \mathtt{e'} \in \mathtt{U} \qquad \mathtt{X} \notin FV(\mathtt{U})}{\Gamma \vdash \mathtt{open\ e\ as\ [X,x]\ in\ e'} \in \mathtt{U}} \qquad \text{(TA-Open)}$$

$$\frac{\Gamma \vdash \mathtt{R} \ll \mathtt{Rec(X)K}}{\Gamma \vdash \mathtt{fold\ [R]\ e} \in \mathtt{EBody(R,R)}{\to}\mathtt{R}} \qquad \text{(TA-Fold)}$$

$$\frac{\Gamma \vdash \mathtt{R} \ll \mathtt{Rec(X)K}}{\Gamma \vdash \mathtt{unfold\ [R]\ e} \in \mathtt{R} \to \mathtt{EBody(R,R)}} \qquad \text{(TA-Unfold)}$$

5.6.3 Theorem [Soundness and completeness of algorithmic typing]: If $\Gamma \vdash \mathtt{e} \in \mathtt{T}$ under the algorithmic definition then $\Gamma \vdash \mathtt{e} \in \mathtt{T}$ under the declarative presentation of typing. If $\Gamma \vdash \mathtt{e} \in \mathtt{T}$ under the declarative presentation of typing then there exists $\mathtt{S}$ such that $\Gamma \vdash \mathtt{e} \in \mathtt{S}$ and, if $\Gamma \vdash \mathtt{e} \in \mathtt{S'}$ declaratively, then $\Gamma \vdash \mathtt{S} <: \mathtt{S'}$.

Proof: The first part proceeds by showing that the algorithmic rules are derivable. The second part uses an induction on algorithmic typing derivations and generation of declarative typing. ∎

5.6.4 Theorem [Subject reduction]: If $\Gamma \vdash \mathtt{e} \in \mathtt{T}$ and $\mathtt{e} \longrightarrow \mathtt{e'}$ then $\Gamma \vdash \mathtt{e'} \in \mathtt{T}$.

Proof: By induction on the length of the reduction sequence establishing $\mathtt{e} \longrightarrow \mathtt{e'}$. At each step, we continue by induction on (declarative) typing derivations.

The argument is now similar to the one for ordinary $F_{\leq}$ because our term formers and reduction rules are identical to $F_{\leq}$. We show the argument here for the case of beta reduction of a type application.

Suppose that the last step in the derivation of $\Gamma \vdash \mathtt{e} \in \mathtt{T}$ was T-TApp, i.e. $\mathtt{e} = \mathtt{(fun(X{<:}T_1)e_0)[T_2]}$ and $\mathtt{T} = \mathtt{[T_2/X]S_2'}$ and $\Gamma \vdash \mathtt{fun(X{<:}S_1)e_0} \in \mathtt{All(X{<:}S_1)S_2'}$ and $\Gamma \vdash \mathtt{T_2} <: \mathtt{S_1}$. The penultimate assumption in turn must have been obtained using T-TAbs followed by (w.l.o.g.) exactly one instance of subsumption. So we may further assume that $\Gamma, \mathtt{X{<:}S_1} \vdash \mathtt{e_0} \in \mathtt{S_2}$ and $\Gamma \vdash \mathtt{All(X{<:}S_1)S_2} <: \mathtt{All(X{<:}S_1)S_2'}$. Now, generation of subtyping yields $\Gamma, \mathtt{X{<:}S_1} \vdash \mathtt{S_2} <: \mathtt{S_2'}$. The result follows using Proposition 5.3.20, subtyping rules, and subsumption. ∎

5.6.5 Remark [Type soundness]: Note that, as in $F_{\leq}$, obviously wrong expressions like `((x,y) z)` or `(fun(x:T)x).2` cannot be typed using the typing rules. Subject reduction then implies that such wrong expressions can never arise during evaluation of a well-typed term.

5.6.6 Remark: Notice that in spite of type soundness $F_{\leq}^{TD}$ is *not* a conservative extension of $F_{\leq}$ with respect to observational equivalence. Indeed, in $F_{\leq}$ the function

$$\mathtt{test} \equiv \mathtt{fun(z{:}All(X{<:}Top{\times}Top)X{\to}X)(z[Int{\times}Int](0,1)).1} \in \mathtt{(All(X{<:}Top{\times}Top)X{\to}X)} \to \mathtt{Int}$$

is observationally equivalent to the constant zero function. Formally, this can be seen using a semantic argument involving a PER model.

In $F_{\leq}^{TD}$ on the other hand these two functions can be distinguished by applying them to an instance of Abadi's `mix` function.[1]

[1]Thanks to Peter O'Hearn and Jon Riecke for pointing this out.

6 Semantics

An important strand of future development for $F_{\leq}^{TD}$ is denotational semantics. We give here a brief sketch of our current ideas.

It appears that we can model the full system $F_{\leq}^{TD}$ using complete uniform pers [Ama91, AC96]. For simplicity here, we omit the recursive types and use ordinary pers. Let *PER* stand for the set of pers (partial equivalence relations) on the natural numbers; see [HP95a] for details on interpretation of ordinary $F_{\leq}$ using pers. The set *TY* of denotations for types is defined inductively as follows.

1. If $R \in PER$ then $\mathtt{Per}(R) \in TY$.
2. $\mathtt{Top} \in TY$.
3. If $A, B \in TY$ then $A{\times}B \in TY$.
4. If $A, B \in TY$ then $\mathtt{!}A{\times}B \in TY$.
5. If $F \in PER \to TY$ then $\mathtt{Some}(F) \in TY$.

Note that the symbols `Top`, `Per`, ×, `!...`×, and `Some` are free constructors of the inductive definition.

The subtyping relation <: $\subseteq TY \times TY$ is defined as follows:

1. A <: `Top` (always).
2. $\mathtt{Per}(R)$ <: $\mathtt{Per}(R')$ if $R \subseteq R'$.
3. $A{\times}B$ <: $A'{\times}B'$ if A <: A' and B <: B'.
4. $\mathtt{!}A{\times}B$ <: $\mathtt{!}A'{\times}B'$ if $A = A'$ and B <: B'.
5. $\mathtt{Some}(F)$ <: $\mathtt{Some}(F')$ if $F(R)$ <: $F'(R)$ for each $R \in PER$.

A function $\overline{} : TY \to PER$ is defined by

$$\begin{array}{lcl}
\overline{\mathtt{Per}(R)} & = & R \\
\overline{\mathtt{Top}} & = & \mathit{Top} \\
\overline{A{\times}B} & = & \overline{A} \times \overline{B} \\
\overline{\mathtt{!}A{\times}B} & = & \overline{A} \times \overline{B} \\
\overline{\mathtt{Some}(F)} & = & \bigsqcup_{R\in PER} \overline{F(R)}
\end{array}$$

Here *Top* denotes the maximal per, × denotes cartesian product of pers, and ⊔ is the symmetric, transitive closure of the set-theoretic union.

Now we can interpret $F_{\leq}^{TD}$ type expressions in an environment which maps variables to elements of *TY* with the understanding that parameters are always mapped to elements of the form $\mathtt{Per}(R)$. The defining clauses are as follows:

$$\begin{array}{lcl}
[\![\mathtt{X}]\!]\eta & = & X \\
[\![\mathtt{Top}]\!]\eta & = & \mathtt{Top} \\
[\![\mathtt{Int}]\!]\eta & = & \mathtt{Per}(\mathit{Int}) \\
[\![\mathtt{T}_1 {\to} \mathtt{T}_2]\!]\eta & = & \mathtt{Per}(\overline{[\![\mathtt{T}_1]\!]\eta} \Rightarrow \overline{[\![\mathtt{T}_2]\!]\eta}) \\
[\![\mathtt{T}_1 {\times} \mathtt{T}_2]\!]\eta & = & [\![\mathtt{T}_1]\!]\eta {\times} [\![\mathtt{T}_2]\!]\eta \\
[\![\mathtt{!}\mathtt{T}_1 {\times} \mathtt{T}_2]\!]\eta & = & \mathtt{!}[\![\mathtt{T}_1]\!]\eta {\times} [\![\mathtt{T}_2]\!]\eta \\
[\![\mathtt{All(X<:T_1)T_2}]\!]\eta & = & \mathtt{Per}(\bigcap_{A <: [\![\mathtt{T}_1]\!]\eta} \overline{[\![\mathtt{T}_2]\!]\eta[\mathtt{X} \mapsto A]}) \\
[\![\mathtt{Some(X)T}]\!]\eta & = & \mathtt{Some}(\lambda R.[\![\mathtt{T}]\!]\eta[\mathtt{X} \mapsto \mathtt{Per}(R)]) \\
[\![\mathtt{T.1}]\!]\eta & = & \begin{cases} A_1, \text{ if } [\![\mathtt{T}]\!]\eta = A_1 {\times} A_2 \\ \text{undefined, otherwise} \end{cases} \\
[\![\mathtt{T.2}]\!]\eta & = & \begin{cases} A_2, \text{ if } [\![\mathtt{T}]\!]\eta = A_1 {\times} A_2 \;\text{ or } [\![\mathtt{T}]\!]\eta = \mathtt{!}A_1 {\times} A_2 \\ \text{undefined otherwise} \end{cases} \\
[\![\mathtt{EBody(U,T)}]\!]\eta & = & \begin{cases} F(\overline{[\![\mathtt{U}]\!]\eta}), \text{ if } [\![\mathtt{T}]\!]\eta = \mathtt{Some}(F) \\ \text{undefined otherwise} \end{cases}
\end{array}$$

Here $\Rightarrow$ denotes function spaces of pers.

This semantics is defined for well-kinded types; η-equal types receive equal meaning, and types standing in the subtype relation are mapped to semantic types standing in the <: relation on TY.

On the level of terms the semantics is as usual; the soundness theorem says that if $\Gamma \vdash \mathtt{e} \in \mathtt{T}$ and η is an appropriate environment then $[\![\mathtt{e}]\!]\eta \in dom(\overline{[\![\mathtt{T}]\!]\eta})$.

Notice that this semantics does not extend to the "ideal system" with bounded existentials from the introduction: If we are allowed to apply a type destructor inside the body of an existential to the bound variable, then the semantic type former `Some` would have to take a function from TY to TY rather than a function from PER to TY as argument; then, however, TY would no longer be inductively defined. It should be possible, though, to replace the inductively defined *set* TY by an appropriately defined *domain* of "semantic type expressions." The details remain to be worked out.

7 Conclusions and Further Work

We have presented a first step towards a general theory of structural subtyping and update by adding type destructors to a version of Kernel Fun with unbounded existentials. The programming examples show that type destructors yield substantially simpler and more readable encodings of object-oriented programming idioms in typed lambda calculus.

Of course, we would like to see the syntactic restrictions on $F_{\leq}^{TD}$ relaxed, while avoiding the bad behavior of the full "ideal system." Apart from the pragmatic solution of living with sound but incomplete checkers, one might look into more refined kinding systems that would retain much of the flexibility of $F_{\leq}$ yet rule out nonterminating type expressions. One promising idea in this direction is based on the observation that, in practice, we only seem to need the type `EBody(U,N)` if `U` is a variable. For example, to type the repack operator

$$\mathtt{repack} \in \mathtt{All(Z<:Some(X)T)} \quad \mathtt{(All(X)All(Y<:T)Y{\to}Y)} \to \quad \mathtt{Z{\to}Z},$$

we only need the equation $\mathtt{Z} =_\eta \mathtt{Some(X)EBody(X,Z)}$, and destructors other than `EBody(X,Z)` do not appear in the course of checking `repack`. The same is true for all the other examples we have checked so far. Therefore, a possible solution might be a system like $F_{\leq}^{TD}$ but with bounded existentials (hence an `EBound` destructor as well as `EBody`) and two kinds of bound variables. The variables of the first kind are allowed to be quantified existentially and to appear as first argument in `EBody` expressions. Only variables of the first kind may be substituted for a variable of the first kind. Variables of the second kind subsume the ones of the first kind and are allowed to be quantified universally, as well as substituted by arbitrary type expressions. We hope that, in this way, one could obtain a proper extension of $F_{\leq}$ which still admits syntax-directed presentations of subtyping and type checking.

Another application of the system with type destructors is as a metalanguage for designing and justifying special-purpose term formers such as the `repack` and polymorphic `unfold` operators. Once designed, these special term formers can be added to ordinary $F_{\leq}$, obtaining the benefits of structural subtyping in particular cases at little cost in terms of meta-theoretic complexity.

References

[AC93] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993. A preliminary version appeared in POPL '91 (pp. 104–118) and as DEC Systems Research Center Research Report number 62, August 1990.

[AC95] Martín Abadi and Luca Cardelli. On subtyping and matching. In *Proceedings ECOOP '95*, pages 145–167, 1995.

[AC96] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[ACV96] Martín Abadi, Luca Cardelli, and Ramesh Viswanathan. An interpretation of objects and object types. In *Principles of Programming Languages*, pages 396–409, 1996.

[Ama91] Roberto M. Amadio. Recursion over realizability structures. *Information and Computation*, 90(2):55–85, 1991.

[BCP97] Kim B. Bruce, Luca Cardelli, and Benjamin C. Pierce. Comparing object encodings. In *Theoretical Aspects of Computer Software (TACS), Sendai, Japan*, September 1997. An earlier version was presented as an invited lecture at the Third International Workshop on Foundations of Object Oriented Languages (FOOL 3), July 1996.

[BPF97] Kim B. Bruce, Leaf Petersen, and Adrian Fiech. Subtyping is not a good "match" for object-oriented languages. In *Proceedings of ECOOP*, volume 1241 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

[Car90] Luca Cardelli. Notes about $F^{\omega}_{<:}$. Unpublished manuscript, October 1990.

[Car92] Luca Cardelli. Extensible records in a pure calculus of subtyping. Research report 81, DEC Systems Research Center, January 1992. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994).

[Car95] Luca Cardelli. Operationally sound update. Talk at *Higher-Order Operational Techniques in Semantics (HOOTS I)*, Cambridge, England, 1995.

[CG92] Pierre-Louis Curien and Giorgio Ghelli. Coherence of subsumption: Minimum typing and type-checking in $F_{\leq}$. *Mathematical Structures in Computer Science*, 2:55–91, 1992. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994).

[CL91] Luca Cardelli and Giuseppe Longo. A semantic basis for Quest. *Journal of Functional Programming*, 1(4):417–458, October 1991. Preliminary version in ACM Conference on Lisp and Functional Programming, June 1990. Also available as DEC SRC Research Report 55, Feb. 1990.

[CM91] Luca Cardelli and John Mitchell. Operations on records. *Mathematical Structures in Computer Science*, 1:3–48, 1991. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994); available as DEC Systems Research Center Research Report #48, August, 1989, and in the proceedings of MFPS '89, Springer LNCS volume 442.

[CMMS94] Luca Cardelli, Simone Martini, John C. Mitchell, and Andre Scedrov. An extension of system F with subtyping. *Information and Computation*, 109(1–2):4–56, 1994. A preliminary version appeared in TACS '91 (Sendai, Japan, pp. 750–770).

[Com94] Adriana B. Compagnoni. Decidability of higher-order subtyping with intersection types. In *Computer Science Logic*, September 1994. Kazimierz, Poland. Springer *Lecture Notes in Computer Science* 933, June 1995. Also available as University of Edinburgh, LFCS technical report ECS-LFCS-94-281, titled "Subtyping in $F^{\omega}_{\wedge}$ is decidable".

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4), December 1985.

[FM96] Kathleen Fisher and John Mitchell. The development of type systems for object-oriented languages. *Theory and Practice of Object Systems*, 1(3):189–220, 1996.

[GLT89] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, 1989.

[HP95a] Martin Hofmann and Benjamin Pierce. Positive subtyping. In *Proceedings of Twenty-Second Annual ACM Symposium on Principles of Programming Languages*, pages 186–197. ACM, January 1995. Full version in *Information and Computation*, volume 126, number 1, April 1996. Also available as University of Edinburgh technical report ECS-LFCS-94-303, September 1994.

[HP95b] Martin Hofmann and Benjamin Pierce. A unifying type-theoretic framework for objects. *Journal of Functional Programming*, 5(4):593–635, October 1995. Previous versions appeared in the Symposium on Theoretical Aspects of Computer Science, 1994, (pages 251–262) and, under the title "An Abstract View of Objects and Subtyping (Preliminary Report)," as University of Edinburgh, LFCS technical report ECS-LFCS-92-226, 1992.

[Pie96] Benjamin C. Pierce. Even simpler type-theoretic foundations for oop. Manuscript (circulated electronically), March 1996.

[Pol96] Erik Poll. Width-subtyping and polymorphic record update. Manuscript, June 1996.

[PS94] Benjamin Pierce and Martin Steffen. Higher-order subtyping. In *IFIP Working Conference on Programming Concepts, Methods and Calculi (PROCOMET)*, 1994. Full version in *Theoretical Computer Science*, vol. 176, no. 1–2, pp. 235–282, 1997 (with a corrigendum in TCS vol. 184 (1997), p. 247).

[PT94] Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–247, April 1994. A preliminary version appeared in Principles of Programming Languages, 1993, and as University of Edinburgh technical report ECS-LFCS-92-225, under the title "Object-Oriented Programming Without Recursive Types".

[Ste98] Martin Steffen. *Polarized Higher-Order Subtyping*. PhD thesis, Universität Erlangen-Nürnberg, 1998. Forthcoming.