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

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 $\mathtt{S}'<:\mathtt{S}$ and $\mathtt{T}'<:\mathtt{T}$. 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 $\mathtt{T} <: \mathtt{U}\times\mathtt{V}$ (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, $F^D_{\leq}$, which imposes some restrictions in order to achieve a tractable metatheory.

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 $\mathtt{All(X<:T)X}\rightarrow\mathtt{X}$ 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 $\mathtt{X} <: \mathtt{T}_1\times\mathtt{T}_2$, then, when `X` is eventually instantiated with a closed type, this type will be a product whose factors are subtypes of $\mathtt{T}_1$ and $\mathtt{T}_2$, respectively. So it should be sound to assume a function

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

such that

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

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 $\mathtt{X} <: \mathtt{Top}\times\mathtt{Top}$ 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:

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

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 $\mathtt{Y} <: \mathtt{Z} \vdash \mathtt{S} <: \mathtt{T}$. Hence (by subsumption) the ordinary unfolding of $\mathtt{x} \in \mathtt{X}$ 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 $F^D_{\leq}$) 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 for 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 <: S1 × S2` for some types `S1` and `S2`.

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:

$$(\mathtt{T_1}\times\mathtt{T_2}).i \;=\; \mathtt{T}_i \qquad \textsc{Beta-Prod}$$

$$\mathtt{T} \;=\; \mathtt{T.1}\times\mathtt{T.2} \qquad \textsc{Eta-Prod}$$

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

```
fun(X<:Top×Top) fun(e:X) fun(e′:X) (e.1,e′.2)
  ∈  All(X<:Top×Top) X→X→X
```

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

```
All(X<:Top×Top) X→X→(Top×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 `T <: !S1×S2`.

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

```
fun(A<:Top) fun(X<:!A×Top)
 fun(e:X) fun(a:A)
 (a,e.2)
∈ All(A<:Top) All(X<:!A×Top) X→A→X.
```

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

```
All(A<:Top) All(X<:!A×Top) X→A→(A×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:

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

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

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

```
EBound(Some(X<:T1)T2)  =  T1
EBody(U,Some(X<:T1)T2) =  [U/X]T2
                     S =  Some(X<:EBound(S))
                           EBody(X,S)
```

(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 ⊢ B ∈ *`. 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 $F^D_{\leq}$

In this paper we take, however, a different approach and describe a restricted system, $F^D_{\leq}$, which does have desirable metatheoretic properties such as decidability of all judgements and type soundness. The most prominent difference between $F^D_{\leq}$ 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 $(T_1 \times T_2).1$ as definitionally equal to 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 $F^D_{\leq}$ 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 $F^{\omega}_{\leq}$

The fragment of $F^D_{\leq}$ without recursive types admits a translation into the higher-order system $F^{\omega}_{\leq}$ [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

$$\texttt{Z <: Some(X) !X}\times\texttt{X}\times\texttt{(X}\rightarrow\texttt{Int)}$$

becomes

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

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

We can thus view (the non-recursive fragment of) $F^D_{\leq}$ as a high-level syntax for such explicit type and operator quantifications. Our experience with a prototype implementation suggests that the use of $F^D_{\leq}$ 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 $F^{\omega}_{\leq}$ with monotone operator subtyping (cf. [Car90, Ste97]). 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 $F^D_{\leq}$ with a definition of its syntax. Section 3 defines the kinding, eta-conversion, subtyping, and typing relations, and summarizes important meta-theoretic properties. In Section 4 we reformulate two familiar encodings of objects—the standard recursive-records model and the simple existential model—in $F^D_{\leq}$. Section 5 sketches a possible denotational semantics for the system. Section 6 offers concluding remarks and some ideas for future work. The detailed development of the metatheory of $F^D_{\leq}$ is omitted here, for the sake of brevity; it can be found in an accompanying technical report [HP97].

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
		$\texttt{T}_1\rightarrow\texttt{T}_2$	function type
		$\texttt{T}_1\times\texttt{T}_2$	product type
		$\texttt{!T}_1\times\texttt{T}_2$	updatable product type
		$\texttt{All(X<:T}_1\texttt{)T}_2$	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 $!\ldots\times$). 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 $\texttt{Some(Y)Y}\times\texttt{Y}$, then `EBody(Int,X)` will become equivalent to $\texttt{Int}\times\texttt{Int}$; 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 $\rightarrow$ 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 $\texttt{T} =_{\eta} \texttt{Dom(T)}\rightarrow\texttt{Cod(T)}$ whenever $\texttt{T<:S}_1\rightarrow\texttt{S}_2$ 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 6).

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 $\Gamma = \Gamma', \texttt{x:T}, \Gamma''$ or $\Gamma = \Gamma', \texttt{X<:T}, \Gamma''$, 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 $\Gamma(\texttt{X}) \stackrel{\text{def}}{=} \texttt{T}$; if `X:*` occurs in Γ then $\Gamma(\texttt{X}) \stackrel{\text{def}}{=} \texttt{*}$. 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 $(\texttt{T}_1 \times \texttt{T}_2)\texttt{.1}$, we need to simplify type expressions when we perform a substitution. To do this we define the substitution `[V/X](T)` of type `V` for `X` in `T` by:

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

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

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

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.,

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

Obviously, these expressions may be undefined.

The term formers of $F^D_{\leq}$ 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
		$\texttt{e}_1$ $\texttt{e}_2$	application
		$\texttt{e}_1\texttt{,e}_2$	pair
		$\texttt{!e}_1\texttt{,e}_2$	updatable pair
		`e.1`	first projection
		`e.2`	second projection
		$\texttt{fun(X<:T}_1\texttt{)e}$	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:

$$\begin{array}{lcl} \texttt{(fun(x:T)e) e}' & \longrightarrow & \texttt{[e}'\texttt{/x]e} \\ \texttt{(fun(X<:T)e)[S]} & \longrightarrow & \texttt{[S/X]e} \\ \texttt{([!]e}_1\texttt{,e}_2\texttt{)}.i & \longrightarrow & \texttt{e}_i \\ \texttt{unfold[R}_1\texttt{] (fold[R}_2\texttt{] e)} & \longrightarrow & \texttt{e} \\ \texttt{open (pack[S,e] as T) as [X,x] in e}' & \longrightarrow & \texttt{[S/X][e/x]e}' \end{array}$$

We sometimes write $\Diamond$ to stand for any of the binary type constructors $!\ldots\times$, $\times$, 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:

$$\begin{array}{rcl} \texttt{K} & ::= & \texttt{Top} \\ & & \texttt{X} \\ & & \texttt{K}_1 \times \texttt{K}_2 \\ & & \texttt{!T}_1 \times \texttt{K}_2 \\ & & \texttt{Some(X)K} \\ & & \texttt{Rec(X)K} \end{array}$$

To state the kinding relation, we need a variant of the substitution operation that treats variables differently depending on where they occur. Suppose that `K` and `L` are kinds, `S` is a type, and `X` is a parameter. Then the substitution of `S` and `K` for `X` in `L` is defined as follows:

$$\begin{array}{lcll} \texttt{[S,K/X]Top} & = & \texttt{Top} & \\ \texttt{[S,K/X]X} & = & \texttt{K} & \\ \texttt{[S,K/X]L}_1 \times \texttt{L}_2 & = & \texttt{[S,K/X]L}_1 \times \texttt{[S,K/X]L}_2 & \\ \texttt{[S,K/X]!T}_1 \times \texttt{L}_2 & = & \texttt{[S/X]!T}_1 \times \texttt{[S,K/X]L}_2 & \\ \texttt{[S,K/X]Some(Y)L}_2 & = & \texttt{Some(Y)[S,K/X]L}_2 & \text{if X} \neq \texttt{Y} \\ \texttt{[S,K/X]Rec(Y)L}_2 & = & \texttt{Rec(Y)[S,K/X]L}_2 & \text{if X} \neq \texttt{Y} \end{array}$$

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

$$\frac{}{\Gamma \vdash \texttt{Top} \ll \texttt{Top}} \quad \text{(K-Top)}$$

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

$$\frac{\Gamma \vdash \texttt{T}_1 \ll \texttt{K}_1 \qquad \Gamma \vdash \texttt{T}_2 \ll \texttt{K}_2}{\Gamma \vdash \texttt{T}_1 \rightarrow \texttt{T}_2 \ll \texttt{Top}} \quad \text{(K-Arr)}$$

$$\frac{\Gamma \vdash \texttt{T}_1 \ll \texttt{K}_1 \qquad \Gamma \vdash \texttt{T}_2 \ll \texttt{K}_2}{\Gamma \vdash \texttt{T}_1 \times \texttt{T}_2 \ll \texttt{K}_1 \times \texttt{K}_2} \quad \text{(K-Prod)}$$

$$\frac{\Gamma \vdash \texttt{T}_1 \ll \texttt{K}_1 \qquad \Gamma \vdash \texttt{T}_2 \ll \texttt{K}_2}{\Gamma \vdash \texttt{!T}_1 \times \texttt{T}_2 \ll \texttt{!T}_1 \times \texttt{K}_2} \quad \text{(K-Upd)}$$

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

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

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

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

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

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

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

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

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

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

We say that T is well-kinded under Γ and write $\Gamma \vdash \mathtt{T} \in *$ if $\Gamma \vdash \mathtt{T} \ll \mathtt{K}$ for some K. 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,

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

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

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

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

plus congruence rules for all the type formers except the bounds of univiersal 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 [HP97]. 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 $\mathtt{T} = \mathtt{!X \times Top}$.)

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 contravariant in the second, unbounded existentials are covariant, and recursive types obey the monotone subtyping rule mentioned in the introduction.

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

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

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

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

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

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

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

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

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.

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

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

$$\frac{\Gamma \vdash \mathtt{S} \ll \mathtt{Rec(X)S_1} \qquad \Gamma \vdash \mathtt{T} \ll \mathtt{Rec(X)T_1} \qquad \Gamma \vdash \mathtt{U} \in * \qquad \Gamma \vdash \mathtt{S} <: \mathtt{T}}{\Gamma \vdash \mathtt{RBody(U,S)} <: \mathtt{RBody(U,T)}} \quad \text{(S-RBody)}$$

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

Notice that the destructors occurring in these rules may be defined ones, so, for example, the following is a valid derivation:

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

Also, note that we might wish for stronger versions of some of these rules, e.g.,

$$\frac{\Gamma \vdash \mathtt{U} <: \mathtt{V} \qquad \Gamma \vdash \mathtt{M} <: \mathtt{N}}{\Gamma \vdash \mathtt{RBody(U,M)} <: \mathtt{RBody(V,N)}}$$

but this would be hard to implement as it introduces a disjunctive case in the subtyping algorithm [HP97].

3.4 Typing

At the level of typing, $F^D_{\leq}$ 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 $\mathtt{X}:*$):

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

The corresponding rule for `pack` is:

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

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

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

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

The typing relation also includes the usual rule of subsumption:

$$\frac{\Gamma \vdash \mathtt{e} \in \mathtt{S} \qquad \Gamma \vdash \mathtt{S} <: \mathtt{T}}{\Gamma \vdash \mathtt{e} \in \mathtt{T}} \quad \text{(T-Subsumption)}$$

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

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

is derivable in $F^D_{\leq}$. If $\mathtt{R} <: \mathtt{Rec(X)T}$ then $\mathtt{R} \ll \mathtt{Rec(X)K}$ by Theorem 3.5.3, 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

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

3.5 Metatheory

Syntax-directed presentations of η-equality, subtyping, and typing allow us to establish the following metatheoretic properties. (The interested reader is referred to the full version [HP97] for proofs.)

3.5.1 Theorem [Substitution]: Let $\mathcal{J}$ be $\mathtt{S} <: \mathtt{T}$ or $\mathtt{S} =_\eta \mathtt{T}$ or $\mathtt{e} \in \mathtt{T}$.

1. If $\Gamma, \mathtt{X}:*, \Delta \vdash \mathcal{J}$ and $\Gamma \vdash \mathtt{U} \in *$ then $\Gamma, \mathtt{[U/X]}\Delta \vdash \mathtt{[U/X]}\mathcal{J}$.
2. If $\Gamma, \mathtt{X}<:\mathtt{V}, \Delta \vdash \mathcal{J}$ and $\Gamma \vdash \mathtt{U} <: \mathtt{V}$ then $\Gamma, \mathtt{[U/X]}\Delta \vdash \mathtt{[U/X]}\mathcal{J}$.

(Notice that this theorem refers to the "β-reducing" substitution operation, and is thus far from trivial.)

3.5.2 Theorem [Decidability]: All judgements of $F^D_{\leq}$ are decidable by syntax-directed procedures.

3.5.3 Theorem [Completeness of Kinding]:

1. If $\Gamma \vdash \mathtt{S} <: \mathtt{T_1}\times\mathtt{T_2}$, then $\Gamma \vdash \mathtt{S} \ll \mathtt{K_1}\times\mathtt{K_2}$ for some $\mathtt{K_1}$ and $\mathtt{K_2}$.
2. If $\Gamma \vdash \mathtt{S} <: \mathtt{!T_1}\times\mathtt{T_2}$, 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 $\Gamma \vdash \mathtt{S} <: \mathtt{Some(X)T}$, then $\Gamma \vdash \mathtt{S} \ll \mathtt{Some(X)K}$ for some $\mathtt{K}$.
4. If $\Gamma \vdash \mathtt{S} <: \mathtt{Rec(X)T}$, then $\Gamma \vdash \mathtt{S} \ll \mathtt{Rec(X)K}$ for some $\mathtt{K}$.

This property fulfills the promise made in the introduction that every subtype of a product is a product, etc.

3.5.4 Theorem [Subject reduction]: If $\Gamma \vdash \mathtt{e} \in \mathtt{T}$ and $\mathtt{e} \longrightarrow \mathtt{e'}$ then $\Gamma \vdash \mathtt{e'} \in \mathtt{T}$.
(The proof of this theorem follows the same lines as in ordinary $F_{\leq}$. The crucial property is substitutivity of subtyping and kinding (Theorem 3.5.1).)

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 implies that such wrong expressions can never arise during evaluation of a well-typed term.

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 $F^D_{\leq}$ with conventional record notation. We introduce the following new syntactic forms:

$$\begin{array}{rcll} \mathtt{A} & ::= & \mathtt{...} & \\ & & \mathtt{\{[!]l_i:T_i;\ ...\}} & \text{record type} \end{array}$$

e	::=	...	
		{[!]l_i=e_i; ...}	record value
		e.l	project ordinary field
		e..l	project updatable field
		e_1 with l:=e_2	update updatable field

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: `{!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 $F^D_{\leq}$, 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 $F^D_{\leq}$. 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 shown in Figure 1.

4.3 Existential Objects

The "simple existential" encoding of objects [PT94, HP95b, etc.] can also be formulated in $F^D_{\leq}$. 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

Y	<:	Cell	given
	=	Rec(X) {get:Int; set:Int→X; bump:X}	by definition
Y	≪	Rec(X) K	by Theorem 3.5.3
Y	$=_\eta$	Rec(Z)RBody(Z,Y)	by Eta-Rec
unfold [Y]	∈	Y → [Y/Z]RBody(Z,Y)	by T-Unfold
	=	Y → RBody(Y,Y)	by defn of substitution
unfold [Y] or	∈	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→Cell; bump:Cell}	by defn of substitution
(unfold [Y] or).bump	∈	Cell	by projection
sendbump	∈	All(Y<:Cell) Y→Y	by abstraction.

Figure 1: Typing of recursive `sendbump`

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
Y	≪	Some(X) K	by Theorem 3.5.3
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×...)
  by S-EBody
  = !Z × ...
  by defn of substitution
EBody(Z,Y) ≪ !Z′ × K₂ (with Z′ $=_\eta$ Z)
  by Theorem 3.5.3
EBody(Z,Y) $=_\eta$!Z′ × EBody(Z,Y).2
  by Eta-Upd
  $=_\eta$!Z × EBody(Z,Y).2
  since $=_\eta$ 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,

!(body.2 body.1).bump, body.2
  ∈ !Z × EBody(Z,Y).2
  $=_\eta$ EBody(Z,Y),

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 Semantics

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

It appears that we can model the full system $F^D_{\leq}$ 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 $!A \times B \in TY$.
5. If $F \in PER \to TY$ then $\mathtt{Some}(F) \in TY$.

Note that the symbols `Top`, `Per`, $\times$, $!\ldots\times$, and `Some` are free constructors of the inductive definition.

The subtyping relation $\mathtt{<:} \subseteq TY \times TY$ 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. $!A \times B$ <: $!A' \times B'$ 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

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

Here Top denotes the maximal per, $\times$ denotes cartesian product of pers, and $\sqcup$ is the symmetric, transitive closure of the set-theoretic union.

Now we can interpret $F^D_{\leq}$ 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:

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

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 $[\![\mathtt{e}]\!]\eta \in \mathit{dom}(\overline{[\![\mathtt{T}]\!]\eta})$.

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.

6 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 $F^D_{\leq}$ 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

```
repack ∈ All(Z<:Some(X)T)
           (All(X)All(Y<:T)Y→Y) →
           Z→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 $F^D_{\leq}$ 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.

[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.

[HP97] Martin Hofmann and Benjamin C. Pierce. Type destructors. Technical report, Computer Science Depart-
ment, Indiana University, 1997. Available through `http://www.cs.indiana.edu/hyplan/pierce.html`.

[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.

[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".

[Ste97] Martin Steffen. *Polarized Higher-Order Subtyping*. PhD thesis, Universität Erlangen-Nürnberg, 1997. Forthcoming.