Operations on Records
(Summary[1])

Luca Cardelli
Digital Equipment Corporation
Systems Research Center

John C. Mitchell
Department of Computer Science
Stanford University

Abstract

We define a simple collection of operations for creating and manipulating record structures, where records are intended as finite associations of values to labels. A second-order type system over these operations supports both subtyping and polymorphism. We provide typechecking algorithms and limited semantic models.

Our approach unifies and extends previous notions of records, bounded quantification, record extension, and parametrization by row-variables. The general aim is to provide foundations for concepts found in object-oriented languages, within a framework based on typed lambda-calculus.

1. Introduction

Object-oriented programming is based on record structures (called *objects*) intended as named collections of values (*attributes*) and functions (*methods*). Collections of objects form *classes*. A *subclass* relation is defined on classes with the intention that methods work "appropriately" on all members belonging to the subclasses of a given class. This property is important in software engineering because it permits after-the-fact extensions of systems by subclasses, without requiring modifications to the systems themselves.

The first object-oriented language, Simula67, and most of the more recent ones (see references) are typed by using simple extensions of the type rules for Pascal-like languages. These extensions mainly involve a notion of *subtyping*. In addition to subtyping, we are interested here in more powerful type systems that smoothly incorporate *parametric polymorphism*.

Type systems for record structures have recently received much attention. They provide foundations for typing in object-oriented languages, data base languages, and their extensions. In [Cardelli 84&88] the basic notions of record types, as intended here, were defined in the context of a first-order type system

[1] *Full paper to appear in Mathematical Structures in Computer Science.*

for fixed-size records. Then Wand [Wand 87] introduced the concept of *row-variables* while trying to solve the type inference problem for records; this led to a system with extensible records and limited second-order typing. His system was later refined and shown to have principal types in [Jategaonkar Mitchell 88], [Rémy 89], and again in [Wand 89]. The resulting system provides a flexible integration of record types and Milner-style type inference [Milner 78].

Meanwhile [Cardelli Wegner 85] defined a full second-order extension of the system with fixed-size records, based on techniques from [Mitchell 84]. In that system, a program can work polymorphically over all the subtypes B of a given record type A, and it can preserve the "unknown" fields (the ones in B but not in A) of record parameters from input to output. However, some natural functions are not expressible. For example, by the nature of fixed-size records there is no way to add a field to a record and preserve all its unknown fields. Less obviously, a function that updates a record field, in the purely applicative sense of making a modified copy of it, is forced to remove all the unknown fields from the result. Imperative update also requires a careful typing analysis.

In this paper we describe a second-order type system that incorporates extensible records and solves the problem of expressing the natural functions mentioned above. We believe this second-order approach makes the presentation of record types more natural. The general idea is to extend a standard second-order (or even higher-order) type system with a notion of subtyping at all types. Record types are then introduced as specialized type constructions with some specialized subtyping rules. These new constructions interact well with the rest of the system. For example, row-variables fall out naturally from second-order type variables, and contravariance of function spaces and universal quantifiers mixes well with record subtyping.

In moving to second-order typing we give up the principal type property of weaker type systems, in exchange for some additional expressiveness. But most importantly for us, we gain some perspective on the space of possible operations on records and record types, unencumbered (at least temporarily) by questions about type inference. Since it is not clear yet where the bounds of expressiveness may lie, this perspective should prove useful for comparisons and further understanding.

The first part of the paper is informal and introduces the main concepts and problems by means of examples. Then we formalize our intuitions by a collection of type rules. In this summary of our work, we briefly describe a normalization procedure for record types, and show soundness of the rules with respect to a simple semantics for the pure calculus of records. Applications and extensions of the basic calculus are described in the full paper.

2. Informal development

Before looking at a formal system, we describe informally the desired operations on records and we justify the rules that are expected to hold. The final formal system is rather subtle, so these explanations should be useful in understanding it.

We also give simple examples of how records and their operations can be used in the context of object-oriented languages.

2.1 Record values

A *record value* is intended to represent, in some intuitive semantic sense, a finite map from labels to values where the values may belong to different types. Syntactically, a record value is a collection of *fields*, where each field is a labeled value. To capture the notion of a map, the labels in a given record must be distinct. Hence the labels can be used to identify the fields, and the fields can be taken to be unordered. This is the notation we use:

$\langle\rangle$	the empty record.
$\langle x=3, y=true\rangle$	a record with two fields, labeled x and y, equivalent to $\langle y=true, x=3\rangle$.

There are three basic operations on record values.

- *Extension* $\langle r \mid x=a\rangle$; adds a field of label x and value a to a record r, provided a field of label x is not already present. (This condition will be enforced statically.) We write $\langle r \mid x=a \mid y=b\rangle$ for $\langle\langle r \mid x=a\rangle \mid y=b\rangle$.
- *Restriction* $r\backslash x$; removes the field of label x, if any, from the record r. We write $r\backslash xy$ for $r\backslash x\backslash y$.
- *Extraction* $r.x$; extracts the value corresponding to the label x from the record r, provided a field having that label is present. (This condition will be enforced statically.)

We have chosen these three operations because they seem to be the fundamental constituents of more complex operations. An alternative, considered in [Wand 87], would be to replace extension and restriction by a single operation that either modifies or adds a field of label x, depending on whether another field of label x is already present. In our system, the extension operation is not required to check whether a new field is already present in a record: its absence is guaranteed statically. The restriction operation has the task of removing unwanted fields and fulfilling that guarantee. This separation of tasks has advantages for efficiency, and for static error detection since fields cannot be overwritten unintentionally by extension alone. Based on a

comparison between the systems of [Wand 87] and [Jategaonkar Mitchell 88], it also seems possible that a reasonable fragment of our language will have a practical type inference algorithm.

Here are some simple examples. The symbol $\leftrightarrow$ (value equivalence) means that two expressions denote the same value.

$\langle\langle\rangle \mid x=3\rangle$	$\leftrightarrow$	$\langle x=3\rangle$	extension	
$\langle\langle x=3\rangle \mid y=true\rangle$	$\leftrightarrow$	$\langle x=3, y=true\rangle$		
$\langle x=3, y=true\rangle \backslash y$	$\leftrightarrow$	$\langle x=3\rangle$	restriction	(cancelling y)
$\langle x=3, y=true\rangle \backslash z$	$\leftrightarrow$	$\langle x=3, y=true\rangle$		(no effect)
$\langle x=3, y=true\rangle.x$	$\leftrightarrow$	3	extraction	

$\langle\langle x=3\rangle \mid x=4\rangle$	invalid extension
$\langle x=3\rangle.y$	invalid extraction

Some useful derived operators can be defined in terms of the ones above.
- *Renaming* $r[x \leftarrow y] =_{\text{def}} \langle r \backslash x \mid y=r.x\rangle$: changes the name of a record field.
- *Overriding* $\langle r \leftarrow x=a\rangle =_{\text{def}} \langle r \backslash x \mid x=a\rangle$: if x is present in r, overriding replaces its value with one of a possibly unrelated type, otherwise extends r (compare with [Wand 89]). Given adequate type restrictions, this can be seen as an updating operator, or a method overriding operator. We write $\langle r \leftarrow x=a \leftarrow y=b\rangle$ for $\langle\langle r \leftarrow x=a\rangle \leftarrow y=b\rangle$.

Obviously, all records can be constructed from the empty record using extension operations. In fact, in the formal presentation of the calculus, we regard the syntax for a record of many fields as an abbreviation for iterated extensions of the empty record, e.g.:

$\langle x=3\rangle$	$=_{\text{def}}$	$\langle\langle\rangle \mid x=3\rangle$
$\langle x=3, y=true\rangle$	$=_{\text{def}}$	$\langle\langle\langle\rangle \mid x=3\rangle \mid y=true\rangle$

This definition allows us to express the fundamental properties of records in terms of combinations of simple operators of fixed arity, as opposed to n-ary operators. Hence, we never have to use schemas with ellipses, such as $\langle x_1=a_1$, ..., $x_n=a_n\rangle$, in our formal treatment.

Since $r \backslash x \leftrightarrow r$ whenever r lacks a field of label x, we can formulate the definition above using any of the following expressions:

$$\langle\langle\rangle \mid x=3 \mid y=true\rangle \;\leftrightarrow\; \langle\langle\langle\rangle \backslash x \mid x=3\rangle \backslash y \mid y=true\rangle \;\leftrightarrow\; \langle\langle\rangle \leftarrow x=3 \leftarrow y=true\rangle$$

The latter forms match better a similar definition for record types, given next.

2.2 Record types

In describing operations on record values we made positive assumptions of the form "a field of label *x must* occur in record *r*" and negative assumptions of the form "a field of label *x must not* occur in record *r*".

These constraints will be verified statically by embedding them in a type system, hence *record types* will convey both positive and negative information. Positive information describes the fields that members of a record type *must* have. (Members may have additional fields.) Negative information describes the fields the members of that type *must not* have. (Members may lack additional fields.)

Note that both positive and negative information expresses constraints, hence increasing either kind of information will lead to smaller sets of values. The smallest amount of information is expressed by the record type with no fields, ⦃⦄, which therefore denotes the collection of all records, since all records have at least no fields and lack at least no fields. This type is called the *total* record type.

⦃⦄	the type of all records. Contains, e.g.: ⟨⟩, ⟨*x*=3⟩.
⦃⦄*x*	the type of all records which lack fields of label *x*. E.g.: ⟨⟩, ⟨*y*=*true*⟩, but not ⟨*x*=3⟩.
⦃*x*:*Int*, *y*:*Bool*⦄	the type of all records which have *at least* fields of labels *x* and *y*, with values of types *Int* and *Bool*. E.g.: ⟨*x*=3, *y*=*true*⟩, ⟨*x*=3, *y*=*true*, *z*="*str*"⟩, but not ⟨*x*=3, *y*=4⟩, ⟨*x*=3⟩.
⦃*x*:*Int*⦄*y*	the type of all records which have *at least* a field of label *x* and type *Int*, and no field of label *y*. E.g. ⟨*x*=3, *z*="*str*"⟩, but not ⟨*x*=3, *y*=*true*⟩.

Hence a record type is characterized by a finite collection of (*positive*) *type fields* (i.e. labeled types) and *negative type fields* (i.e. labels)[2]. We often simply say "fields" for "type fields". The positive fields must have distinct labels and are unordered. Negative fields are also unordered. We have assumed so far that types are normalized so that positive and negative labels are distinct, otherwise positive and negative fields may cancel, as described shortly.

[2]In this section we consider only *ground* record types, i.e., those containing no record type variables.

As with record values, we have three basic operations on record types.

• *Extension* $\langle\!\langle R \mid x{:}A \rangle\!\rangle$: This type denotes the collection obtained from R by adding x fields with values in A in all possible ways (provided that none of the elements of R has x fields). More precisely, this is the collection of those records $(r \mid x{=}a)$ such that r is in R and a is in A, provided that a positive type field x is not already present in R. (This condition will be enforced statically.) We sometimes write $\langle\!\langle R \mid x{:}A \mid y{:}B \rangle\!\rangle$ for $\langle\!\langle \langle\!\langle R \mid x{:}A \rangle\!\rangle \mid y{:}B \rangle\!\rangle$.

• *Restriction* $R \backslash x$: this type denotes the collection obtained from R by removing the field x (if any) from all its elements. More precisely, this is the collection of those records $r \backslash x$ such that r is in R. We write $R \backslash xy$ for $R \backslash x \backslash y$.

• *Extraction* $R.x$: this type denotes the type associated with label x in R, provided R has such a positive field. (This condition will be enforced statically.)

Again, derived operators can be defined in terms of the ones above.

• *Renaming* $R[x \leftarrow y] =_{\text{def}} \langle\!\langle R \backslash x \mid y{=}R.x \rangle\!\rangle$: changes the name of a record type field.

• *Overriding* $\langle\!\langle R \leftarrow x{:}A \rangle\!\rangle =_{\text{def}} \langle\!\langle R \backslash x \mid x{:}A \rangle\!\rangle$: if a type field x is present in R, overriding replaces it with a field x of type A, otherwise extends R. Given adequate type restrictions, this can be used to override a method type in a class signature (i.e. record type) with a more specialized one, to produce a subclass signature.

The crucial formal difference between these operators on types and the similar ones on values is that type restrictions do not cancel as easily, for example: $\langle\!\langle \rangle\!\rangle \backslash y \neq \langle\!\langle \rangle\!\rangle$, $\langle\!\langle x{:}A \rangle\!\rangle \backslash y \neq \langle\!\langle x{:}A \rangle\!\rangle$, etc., since $\langle\!\langle \rangle\!\rangle \backslash y$ is a smaller set than $\langle\!\langle \rangle\!\rangle$. As a consequence, one must always make a type restriction before making a type extension, as can be seen in the examples below, because the extension operator needs proof that the extension label is missing. The symbol $\leftrightarrow$ (type equivalence) means also that two type expressions denote the same type.

$\langle\!\langle \langle\!\langle \rangle\!\rangle \backslash x \mid x{:}Int \rangle\!\rangle$	$\leftrightarrow$	$\langle\!\langle x{:}Int \rangle\!\rangle$	extension	
$\langle\!\langle \langle\!\langle x{:}Int \rangle\!\rangle \backslash y \mid y{:}Bool \rangle\!\rangle$	$\leftrightarrow$	$\langle\!\langle x{:}Int, y{:}Bool \rangle\!\rangle$		
$\langle\!\langle x{:}Int, y{:}Bool \rangle\!\rangle \backslash y$	$\leftrightarrow$	$\langle\!\langle x{:}Int \rangle\!\rangle \backslash y$	restriction	(cancelling y)
$\langle\!\langle x{:}Int, y{:}Bool \rangle\!\rangle \backslash z$	$\leftrightarrow$	$\langle\!\langle x{:}Int, y{:}Bool \rangle\!\rangle \backslash z$		(no effect on x,y)
$\langle\!\langle x{:}Int, y{:}Bool \rangle\!\rangle.x$	$\leftrightarrow$	Int	extraction	
$\langle\!\langle \langle\!\langle \rangle\!\rangle \mid x{:}Int \rangle\!\rangle$		invalid extension		
$\langle\!\langle \langle\!\langle x{:}Int \rangle\!\rangle \mid x{:}Int \rangle\!\rangle$		invalid extension		
$\langle\!\langle x{:}Int \rangle\!\rangle.y$		invalid extraction		

It helps to read these examples in terms of the collections they represent. For

example, the first example for restriction says that if we take the collection of records that have x and y (and possibly more) fields, and remove the y field from all the elements in the collection, then we obtain the collection of records that have an x field (and possibly more fields) but no y field. In particular, we do not obtain the collection of records that have x and possibly more fields, because those would include y.

The way positive and negative information is formally manipulated is easier to understand if we regard record types as abbreviations, as we did for record values, e.g.:

$$\begin{array}{lll} \langle\!\langle x{:}Int \rangle\!\rangle & =_{\text{def}} & \langle\!\langle \langle\!\langle \rangle\!\rangle \backslash x \mid x{:}Int \rangle\!\rangle \\ \langle\!\langle x{:}Int, y{:}Bool \rangle\!\rangle & =_{\text{def}} & \langle\!\langle \langle\!\langle \langle\!\langle \rangle\!\rangle \backslash x \mid x{:}Int \rangle\!\rangle \backslash y \mid y{:}Bool \rangle\!\rangle \end{array}$$

Then, when considering $\langle\!\langle y{:}Bool \rangle\!\rangle \backslash y$ we actually have the expansion $\langle\!\langle \langle\!\langle \rangle\!\rangle \backslash y \mid y{:}Bool \rangle\!\rangle \backslash y$. If we allow the outside positive and negative y labels to cancel, we are still left with $\langle\!\langle \rangle\!\rangle \backslash y$. In other words, the inner y restriction reminds us that y fields have been eliminated.

2.3 Record value variables

Now that we have a first understanding of record types, we can introduce record value variables which are declared to have some record type. For example, $r{:}\langle\!\langle \rangle\!\rangle \backslash y$ means that r must not have a field y, and $r{:}\langle\!\langle x{:}A \rangle\!\rangle$ means that r must have a field x of type A. The well-formed record expressions can now be formulated more precisely:

$$\begin{array}{rl} \langle r \mid x{=}a \rangle & \text{where } r{:}\langle\!\langle \rangle\!\rangle \backslash x \\ r \backslash x & \text{where } r{:}\langle\!\langle \rangle\!\rangle \\ r.x & \text{where } r{:}\langle\!\langle x{:}A \rangle\!\rangle \text{ for some } A \end{array}$$

Record value variables can now be used to write function abstractions. Here we have a function that increments a field of a record, and adds another field to it:

$$\begin{array}{l} \mathit{let}\ f(r{:}\ \langle\!\langle x{:}Int \rangle\!\rangle \backslash y) : \langle\!\langle x{:}Int, y{:}Int \rangle\!\rangle = \\ \quad \langle r \leftarrow x{=}r.x{+}1 \mid y{=}0 \rangle \end{array}$$

This function requires an argument with a field x and no field y; it has type:

$$f : \langle\!\langle x{:}Int \rangle\!\rangle \backslash y \rightarrow \langle\!\langle x{:}Int, y{:}Int \rangle\!\rangle$$

and can be used as follows:

$$f(\langle x{=}3\rangle) \quad \leftrightarrow \quad \langle x{=}4, y{=}0\rangle : \langle\!\langle x{:}Int, y{:}Int\rangle\!\rangle$$
$$f(\langle x{=}3, z{=}true\rangle) \quad \leftrightarrow \quad \langle x{=}4, y{=}0, z{=}true\rangle : \langle\!\langle x{:}Int, y{:}Int\rangle\!\rangle$$

The first application uses the non-trivial fact that $\langle x{=}3\rangle : \langle\!\langle x{:}Int\rangle\!\rangle \backslash y$. We could also have matched the parameter type precisely by $f(\langle x{=}3\rangle \backslash y)$, which is of course equivalent. The second application is noticeable for several reasons. First, it uses the non-trivial fact that $\langle x{=}3, z{=}true\rangle : \langle\!\langle x{:}Int\rangle\!\rangle \backslash y$. Second, the "extra" field z is preserved in the result value, because of the way f is defined. Third, the "extra" field z is not preserved in the result type, because f has a fixed result type; we shall come back to this problem.

2.4 Record type variables

In the previous section we introduced record value variables, and we used record types to impose restrictions on the values which could be bound to such variables. Now we want to introduce record type variables in order to write programs that are polymorphic over a collection of record types. We similarly need to express restrictions on the admissible types that these variables can be bound to; these restrictions are written as subtype specifications.

To write subtype specifications, we use a predicate $A{<:}B$ meaning that A is a *subtype* of B: in other words, every value of A is also a value of B. The typing rule based on this condition is called *subsumption*, and will play a central role in the formal system.

Using subtype assumptions, we can better formulate the restrictions on the record type operators:

$$\begin{array}{rl} \langle\!\langle R \mid x{:}A\rangle\!\rangle & \text{where } R <: \langle\!\langle\rangle\!\rangle \backslash x \\ R \backslash x & \text{where } R <: \langle\!\langle\rangle\!\rangle \\ R.x & \text{where } R <: \langle\!\langle x{:}A\rangle\!\rangle \text{ for some } A \end{array}$$

We may now write a polymorphic version of the function f of the previous section:

$$\begin{array}{l} \mathit{let}\ f(R{<:}\langle\!\langle x{:}Int\rangle\!\rangle \backslash y)(r{:}R) : \langle\!\langle R \mid y{:}Int\rangle\!\rangle = \\ \quad \langle r \leftarrow x{=}r.x{+}1 \mid y{=}0\rangle \end{array}$$

This function expects first a type parameter R which must be a subtype of $\langle\!\langle x{:}Int\rangle\!\rangle \backslash y$, and then an actual value parameter of type R. An example application is:

$$\begin{array}{l} f(\langle\!\langle x{:}Int, z{:}Bool\rangle\!\rangle \backslash y)(\langle x{=}3, z{=}true\rangle) \leftrightarrow \\ \quad \langle x{=}4, y{=}0, z{=}true\rangle : \langle\!\langle x{:}Int, y{:}Int, z{:}Bool\rangle\!\rangle \end{array}$$

First, note that R is bound to $\langle\!\langle x{:}Int, z{:}Bool \rangle\!\rangle \backslash y$, which is a subtype of $\langle\!\langle x{:}Int \rangle\!\rangle \backslash y$ as required. Second, $\langle x{=}3, z{=}true \rangle$ has type $\langle\!\langle x{:}Int, z{:}Bool \rangle\!\rangle \backslash y$ as required. Third, the result type, obtained by instantiating R, is $\langle\!\langle \langle\!\langle x{:}Int, z{:}Bool \rangle\!\rangle \backslash y \mid y{:}Int \rangle\!\rangle$, which is the same as $\langle\!\langle x{:}Int, y{:}Int, z{:}Bool \rangle\!\rangle$ by definition. Finally, note that the "extra" field z has not been forgotten in the result type this time, because all the "extra" fields are carried over from input to output type by the type variable R. This is the advantage of writing f in polymorphic style.

What is the type of f then? We cannot write this type with simple function arrows, because we have a free variable R to bind. Moreover, we want to mark the precise location where this binding occurs, because this permits more types to be expressed. Hence, we use an explicit *bounded universal quantifier*:

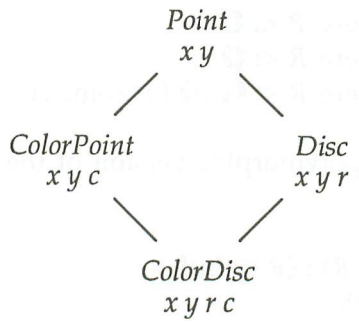
$$f : \forall (R <: \langle\!\langle x{:}Int \rangle\!\rangle \backslash y)\ R \to \langle\!\langle R \mid y{:}Int \rangle\!\rangle$$

This reads rather naturally: "for all types R which are subtypes of $\langle\!\langle x{:}Int \rangle\!\rangle \backslash y$, f is a function form R to $\langle\!\langle R \mid y{:}Int \rangle\!\rangle$". (The scope of a quantifier extends to the right as much as possible.)

2.5 Subtype hierarchies

Our operations on record types and record values make it easy to define new types and values by *reusing* previously defined types and values.

For example, we want to express the subtype hierarchy shown in the diagram below, where various entities can have a combination of coordinates x and y, radius r, and color c.



First, we could define each type independently:

let Point = $\langle\!\langle x{:}Real, y{:}Real \rangle\!\rangle$
let ColorPoint = $\langle\!\langle x{:}Real, y{:}Real, c{:}Color \rangle\!\rangle$

let Disc = ⟪*x*:*Real*, *y*:*Real*, *r*:*Real*⟫
let ColorDisc = ⟪*x*:*Real*, *y*:*Real*, *r*:*Real*, *c*:*Color*⟫

But these explicit definitions do not scale up easily to large hierarchies; it is much more convenient to define each type in terms of previous ones, e.g:

let Point = ⟪*x*:*Real*, *y*:*Real*⟫
let ColorPoint = ⟪*Point* ← *c*:*Color*⟫
let Disc = ⟪*Point* ← *r*:*Real*⟫
let ColorDisc = ⟪*ColorPoint* ← *r*:*Real*⟫

Note that ⟪*Point* | *c*:*Color*⟫ would not be well-formed here, since members of *Point* may have a *c* label.

Similarly, record values can be defined by reusing available values:

let p:*Point* = ⟨*x*=3, *y*=4⟩
let cp:*ColorPoint* = ⟨*p* ← *c*=*green*⟩
let cd:*ColorDisc* = ⟨*cp* ← *r*=1⟩
let d:*Disc* = *cd**c*

We should notice here that the subtyping relation depends only on the structure of the types, and not on how the types are named or constructed. Similarly, record values belong to record types uniquely based on their structure, independently of how they are declared or constructed.

Another observation, which we already made in a more abstract context, is that *Point**r* <: *Point* since *Point* does not contain *r*, but *Point**y* is incomparable with *Point* since *Point* requires *y*:*Int* while *Point**y* forbids it.

2.6 The update problem

The type system for records we have described in the previous sections was initially motivated by a single example which involves typing an update function. Here updating is intended in the functional sense of creating a copy of a record with a modified field, but the discussion is also relevant to imperative updating.

The problem is to define a function that updates a field of a record and returns the new record; the type of this function should be such that when an argument of the function has a subtype of the expected input type, the result has a related subtype. That is, no type information regarding additional fields should be lost in updating. (We have already seen that bounded quantification can be useful in this respect.)

It is pretty clear what the body of such a function should look like; for

example for an input r and a boolean field b which has to be negated, we would write:

$$\langle r \leftarrow b{=}not(r.b)\rangle \qquad \text{(an abbreviation for } \langle r \backslash b \mid b{=}not(r.b)\rangle \text{)}$$

The overriding operator here preserves the additional fields of r.

One might expect the following typing, which seems to preserve subtype information as desired:

$$\begin{array}{l} let\ update(R{<:}\{\!\{b{:}Bool\}\!\})(r{:}R){:}\ R = \\ \quad \langle r \leftarrow b{=}not(r.b)\rangle \end{array}$$

In words, we expect *update* to be a function from R to R, for any subtype R of $\{\!\{b{:}Bool\}\!\}$. But this typing is not derivable from our rules and, worse, it is semantically unsound. To see this, assume we have a type $True <: Bool$ with unique element *true*, as follows[3]:

$$\begin{array}{l} true : True <: Bool \\ not : Bool \to Bool \qquad \text{(alternatively, } not : \forall(A{<:}Bool)A{\to}Bool\text{)} \end{array}$$

$$update(\{\!\{b{:}True\}\!\})(\langle b{=}true\rangle) \ \leftrightarrow\ \langle b{=}false\rangle : \{\!\{b{:}True\}\!\}$$

This use of *update* produces an obviously incorrect result type. In general, a function with result type R has a fixed range; it cannot restrict its output to an arbitrary subtype of R, even when this subtype is given as a parameter.

To avoid this problem, we must update the result type as well as the result. The correct typing comes naturally from typechecking the body of *update* according to the rules for each construct involved; note how the shape of the result type matches the shape of the body of the function:

$$\begin{array}{l} let\ update(R{<:}\{\!\{b{:}Bool\}\!\})(r{:}R){:}\ \{\!\{R{\leftarrow}b{:}Bool\}\!\} = \\ \quad \langle r \leftarrow b{=}not(r.b)\rangle \end{array}$$

$$\begin{array}{l} update(\{\!\{b{:}True\}\!\})(\langle b{=}true\rangle) \ \leftrightarrow \\ \quad \langle b{=}false\rangle :\ (\{\!\{\{\!\{b{:}True\}\!\}{\leftarrow}b{:}Bool\}\!\} \leftrightarrow \{\!\{b{:}Bool\}\!\}) \end{array}$$

The outcome is that the overriding operator on types, which involves manipulation of negative information, is necessary to express the type of update functions. Bounded quantification by itself is not sufficient.

The type $\forall(B{<:}A)\ B \to B$ turns out to contain only the identity function on

[3]Although the singleton type *True* may seem artificial, this argument can be reformulated with any proper inclusion between two types.

A in many natural semantic models, such as [Bruce Longo 88]. For example take A=*Int* and let the subranges $[n..m]$ be subtypes of *Int*. Then any function of type $\forall(B<:Int)\ B \to B$ can be instantiated to $[n..n] \to [n..n]$, hence it must be the identity on $[n..n]$ for any n, and hence the identity over all of *Int*.

A further complication manifests itself when updating acts deep in a structure, because then we have to preserve type information with subtyping occurring at multiple levels. Here is the body of a function that negates the *s.a.b* field of a record *s* of type $\langle\!\langle a{:}\langle\!\langle b{:}Bool\rangle\!\rangle\rangle\!\rangle$:

$$\langle s \leftarrow a = \langle s.a \leftarrow b = not(s.a.b)\rangle\rangle$$

The following is a correct typing which does not lose information on subtypes (simpler typings would). Here we need to introduce an additional type parameter in order to use two type variables in the result type and to avoid two possible ways of losing type information:

$$\begin{aligned}&let\ deepUpdate(R<:\langle\!\langle b{:}Bool\rangle\!\rangle)(S<:\langle\!\langle a{:}R\rangle\!\rangle)(s{:}S){:}\ \langle\!\langle S \leftarrow a{:}\langle\!\langle R \leftarrow b{:}Bool\rangle\!\rangle\rangle\!\rangle =\\ &\quad \langle s \leftarrow a = \langle s.a \leftarrow b = not(s.a.b)\rangle\rangle\end{aligned}$$

Of course this is rather clumsy; we need one additional type parameter for each additional depth level of updating. Fortunately, we can avoid the extra type parameters by using *extraction* types *S.a*. Again, the following typing comes naturally from typechecking the body of *deepUpdate* according to the rules for each construct:

$$\begin{aligned}&let\ deepUpdate(S<:\langle\!\langle a{:}\langle\!\langle b{:}Bool\rangle\!\rangle\rangle\!\rangle)(s{:}S){:}\ \langle\!\langle S \leftarrow a{:}\langle\!\langle S.a \leftarrow b{:}Bool\rangle\!\rangle\rangle\!\rangle =\\ &\quad \langle s \leftarrow a = \langle s.a \leftarrow b = not(s.a.b)\rangle\rangle\end{aligned}$$

The output type is still complex (it could be inferred) but the input is more natural. Here is a use of this function:

$$\begin{aligned}&deepUpdate(\langle\!\langle a{:}\langle\!\langle b{:}True, c{:}C\rangle\!\rangle, d{:}D\rangle\!\rangle)(\langle a = \langle b = true, c{:}v\rangle, d{:}w\rangle) \ \leftrightarrow\\ &\quad \langle a = \langle b = true, c{:}v\rangle, d{:}w\rangle : \langle\!\langle a{:}\langle\!\langle b{:}Bool, c{:}C\rangle\!\rangle, d{:}D\rangle\!\rangle\end{aligned}$$

Here we have provided an argument type that is a subtype of $\langle\!\langle a{:}\langle\!\langle b{:}Bool\rangle\!\rangle\rangle\!\rangle$ in "all possible ways".

Finally, we should remark that the complexity of the update problem seems to manifests itself only in the functional case, while simpler solutions are available in the imperative case. Simpler type systems for records, such as the one in [Cardelli Wegner 85], may be adequate for imperative languages when

properly extended with imperative constructs, as sketched below.

The imperative updating operator := has the additional constraint that the new record should have the same type as the old record, since intuitively updating is done "in place". This requirement produces something very similar to the typing we have initially shown to be unsound. Here assignable fields are identified by *var*:

$$\begin{array}{l} \mathit{let}\ \mathit{update}(R{<:}\langle\!\langle \mathit{var}\ b{:}\mathit{Bool}\rangle\!\rangle)(r{:}R){:}\ R = \\ \quad r.b := \mathit{not}(r.b) \end{array}$$

Soundness is then recovered by requiring that assignable fields be both covariant and contravariant. Hence, $\mathit{True} <: \mathit{Bool}$ does not imply $\langle\!\langle \mathit{var}\ b{:}\mathit{True}\rangle\!\rangle <: \langle\!\langle \mathit{var}\ b{:}\mathit{Bool}\rangle\!\rangle$, thereby blocking the counterexamples to soundness.

Imperative update, with the natural requirement of not changing the type of a record, leads to simpler typing. However, this approach does not completely solve the problem we have discussed in this section. Imperative update alone does not provide the functionality of polymorphically extending existing records; when this is added, all the problems discussed above about functional update resurface.

3. Formal development

Now that we have acquired some intuitions, we can discuss the formal type inference rules in detail. We first define judgment forms and environment structures. Then we look at inference rules individually, and we analyze their properties. Finally, we provide a set-theoretical semantics for the pure calculus of records.

3.1 Judgments and inferences

A *judgment* is an inductively defined predicate between environments, value terms and type terms. The following judgments are used in formalizing our system:

$\vdash E\ \mathit{env}$	E is an environment
$E \vdash A\ \mathit{type}$	A is a type
$E \vdash A <: B$	A is a subtype of B
$E \vdash a : A$	a has type A
$E \vdash A \leftrightarrow B$	equivalent types
$E \vdash a \leftrightarrow b : A$	equivalent values of type A

The formal system is given by a set of *inference rules* below, each expressed as a finite set of *antecedent* judgments and side conditions (above a horizontal line) and a single *conclusion* judgment (below the line). Most inference rules are actually *rule schemas*, where meta-variables must be instantiated to obtain concrete inferences. For typographical reasons, we write the side conditions for these schemas as part of the antecedent.

3.2 Environments

An environment E is a finite sequence of (a) unconstrained type variables, (b) type variables constrained to be subtypes of a given type, and (c) value variables associated with their type.

We use $dom(E)$ for the set of type and value variables defined in an environment.

$$\text{(ENV1)}\ \frac{}{\vdash \emptyset\ env} \qquad \text{(ENV2)}\ \frac{X \notin dom(E)}{\vdash E, X\ env} \qquad \text{(ENV3)}\ \frac{E \vdash A\ type \quad X \notin dom(E)}{\vdash E, X{<:}A\ env} \qquad \text{(ENV4)}\ \frac{E \vdash A\ type \quad x \notin dom(E)}{\vdash E, x{:}A\ env}$$

Hence, a legal environment is obtained by starting with the empty environment $\emptyset$ and extending it with a finite set of *assumptions* for type and value variables. Note that the assumptions involve distinct variables; we could perhaps allow multiple assumptions (e.g., $\emptyset, X{<:}A, X{<:}B$) but this would push us into the more general discipline of *conjunctive types*.

Assumptions about variables can then be extracted from well-formed environments:

$$\text{(VAR1)}\ \frac{\vdash E,X,E'\ env}{E,X,E' \vdash X\ type} \qquad \text{(VAR2)}\ \frac{\vdash E,X{<:}A,E'\ env}{E,X{<:}A,E' \vdash X\ type} \qquad \text{(VAR3)}\ \frac{\vdash E,X{<:}A,E'\ env}{E,X{<:}A,E' \vdash X{<:}A} \qquad \text{(VAR4)}\ \frac{\vdash E,x{:}A,E'\ env}{E,x{:}A,E' \vdash x{:}A}$$

All legal inferences take place in (well-formed) environments. All judgments are recursively defined in terms of other judgments. For example, above we have used the typing judgment $E \vdash A\ type$ in constructing environments; vice versa, well-formed environments are involved in constructing types.

We now consider the remaining judgments in turn.

3.3 Record type formation

The following collection of rules determines when record types are well-formed. There is some interdependence between this section and the following

ones, since equivalence rules have assumption that involve subtyping, which is discussed later. Fortunately, these assumptions are fairly simple, so a full understanding of the subtype relation is not required at this point.

$$\text{(F1)}\ \frac{\vdash E\ env}{E \vdash \langle\!\langle\rangle\!\rangle\ type} \qquad \text{(F2)}\ \frac{E \vdash R <: \langle\!\langle\rangle\!\rangle \backslash x \quad E \vdash A\ type}{E \vdash \langle\!\langle R \mid x{:}A \rangle\!\rangle\ type} \qquad \text{(F3)}\ \frac{E \vdash R <: \langle\!\langle\rangle\!\rangle}{E \vdash R \backslash x\ type} \qquad \text{(F4)}\ \frac{E \vdash R <: \langle\!\langle S \mid x{:}A \rangle\!\rangle <: \langle\!\langle\rangle\!\rangle}{E \vdash R.x\ type}$$

As shown above, and already discussed informally, the legal record types are: the type of all records, $\langle\!\langle\rangle\!\rangle$; a record type variable X, (because of (VAR3) in the previous section); an extension $\langle\!\langle R \mid x{:}A \rangle\!\rangle$ of a record type R, provided R does not have x; and a restriction $R \backslash x$ of a record type R. Moreover, extracting a component $R.x$ of a record type R that has a label x, produces a legal type.

In general, if R does not have x, then R will be a subtype of the type $\langle\!\langle\rangle\!\rangle \backslash x$ of all records without x. This explains the hypothesis of rule (F2). In rule (F4) we use $R <: \langle\!\langle S \mid x{:}A \rangle\!\rangle$ to guarantee that every record in R has an x field.

3.4 Record type equivalence

When are two record types equivalent? We discuss here the formal rules for answering such a question. Type equivalence, as a relation, is reflexive (over well-formed expressions), symmetric, and transitive; it is denoted by the symbol $\leftrightarrow$. Substituting two equivalent types in a third type should produce an equivalent result; this is called the *congruence* property, and requires a number of rules to be fully formalized. We now consider, by cases, the equivalence of extended, restricted and extracted record types.

Two extended record types are equivalent if we can reorder their fields to make them identical (or, recursively, equivalent). This simple fact is expressed by the following rule. A number of applications of this rule, and of the congruence property, may be necessary to adequately reorder the fields of a record type.

$$\text{(TE1)}\ \frac{E \vdash R <: \langle\!\langle\rangle\!\rangle \backslash x y \quad E \vdash A,B\ type \quad x \neq y}{E \vdash \langle\!\langle \langle\!\langle R \mid x{:}A \rangle\!\rangle \mid y{:}B \rangle\!\rangle \leftrightarrow \langle\!\langle \langle\!\langle R \mid y{:}B \rangle\!\rangle \mid x{:}A \rangle\!\rangle}$$

Similarly, we can reorder restrictions. Moreover, a double restriction $R \backslash x x$ reduces to $R \backslash x$. This fact is expressed in slightly more general form below, since the assumption that R does not have x is sufficient to deduce that $R \backslash x$ is the same as R:

(TE2)
$$\frac{E \vdash R <: \langle\!\langle\rangle\!\rangle \backslash x}{E \vdash R \backslash x \leftrightarrow R}$$

(TE3)
$$\frac{E \vdash R <: \langle\!\langle\rangle\!\rangle}{E \vdash R \backslash xy \leftrightarrow R \backslash yx}$$

The most interesting rules concern the distribution of restriction over extension. An outside restriction and inner extension of the same variable can cancel each other. Otherwise, a restriction can be pushed inside or outside of an extension of a different variable.

(TE5)
$$\frac{E \vdash R <: \langle\!\langle\rangle\!\rangle \backslash x \quad E \vdash A\ type}{E \vdash \langle\!\langle R \mid x{:}A \rangle\!\rangle \backslash x \leftrightarrow R}$$

(TE6)
$$\frac{E \vdash R <: \langle\!\langle\rangle\!\rangle \backslash x \quad E \vdash A\ type \quad x \neq y}{E \vdash \langle\!\langle R \mid x{:}A \rangle\!\rangle \backslash y \leftrightarrow \langle\!\langle R \backslash y \mid x{:}A \rangle\!\rangle}$$

Note however that in a situation like $\langle\!\langle R \backslash x \mid x{:}A \rangle\!\rangle$ no cancellation or swap can occur. The inner restriction may be needed to guarantee that the extension is sensible, and so neither is redundant.

Finally, a record extraction is equivalent to the extracted type:

(TE7)
$$\frac{E \vdash R <: \langle\!\langle\rangle\!\rangle \backslash x \quad E \vdash A\ type}{E \vdash \langle\!\langle R \mid x{:}A \rangle\!\rangle.x \leftrightarrow A}$$

(TE8)
$$\frac{E \vdash R <: \langle\!\langle S \mid y{:}B \rangle\!\rangle \backslash x <: \langle\!\langle\rangle\!\rangle \quad E \vdash A\ type \quad x \neq y}{E \vdash \langle\!\langle R \mid x{:}A \rangle\!\rangle.y \leftrightarrow R.y}$$

(TE4)
$$\frac{E \vdash R <: \langle\!\langle S \mid y{:}B \rangle\!\rangle <: \langle\!\langle\rangle\!\rangle \quad x \neq y}{E \vdash R \backslash x.y \leftrightarrow R.y}$$

These equivalence rules can be given a direction and interpreted as rewrite rules producing a normal form for record types; normalization is investigated in a later section.

3.5 Record subtyping

We have seen that subtyping is central to the notion of abstracting over record type variables, and we have intuitively justified some of the valid subtype assertions. In this section we take a more rigorous look at the subtype relation.

Subtyping should at least be a pre-order: a reflexive and transitive relation. Given a substitutive type equivalence relation $\leftrightarrow$, such as the one discussed in the previous section, we require:

(G1)
$$\frac{E \vdash A \leftrightarrow B}{E \vdash A <: B}$$

(G2)
$$\frac{E \vdash A <: B \quad E \vdash B <: C}{E \vdash A <: C}$$

Reflexivity is a special case of (G1).

It would be natural to require subtyping to be anti-symmetric, hence obtaining a partial order. A reasonable semantics of subtyping will in fact construct such a partial order. However, it might be too strong to require anti-symmetry as a type rule. In some systems anti-symmetry may introduce obscure ways of proving type equivalence, while in other systems it may be provable from the other rules. Moreover, anti-symmetry does not seem very useful for typechecking, hence we do not include it.

The basic intuition about subtyping is that it behaves much like the subset relation; this is expressed by the *subsumption* rule, which claims that if $A<:B$ and a is an element of A, then a is also an element of B.

(G3)
$$\frac{E \vdash a{:}A \quad E \vdash A <: B}{E \vdash a : B}$$

We feel strongly that subsumption should be included in the type system, since this rule gives object-oriented programming much of its flavor. One should not be satisfied, for programming purposes, with emulating subsumption by explicit coercions. The latter technique is interesting and adequate for providing semantics to a language with subsumption [Breazu-Tannen Coquand Gunter Scedrov 89] [Curien Ghelli 89], but even then it would seem more satisfactory to exhibit a model that satisfies subsumption directly.

Combining (G1) and (G3) we obtain another standard type rule:

$$\frac{E \vdash a{:}A \quad E \vdash A \leftrightarrow B}{E \vdash a : B}$$

This rule is normally taken as primitive, but here it is derived.

We are now ready to talk about subtyping between record types. It helps if we break this problem into pieces and ask what are the subtypes of: (1) the total record type $\langle\!\langle\rangle\!\rangle$, (2) an extended record type $\langle\!\langle R \mid x{:}A \rangle\!\rangle$, (3) a restricted record type $R \backslash x$, and (4) a record type extraction $R.x$.

Case (1). Every record type should be a subtype of the total record type. Hence, we have three subcases: (1a) the total record type is of course a subtype of itself, and this is simply a consequence of (G1); (1b) any well-formed extended record type is a subtype of $\langle\!\langle\rangle\!\rangle$; and (1c) any well-formed restricted record type is a subtype of $\langle\!\langle\rangle\!\rangle$. Hence we have the following rules corresponding to 1b and 1c respectively:

(S1)
$$\frac{E \vdash R <: \langle\!\langle\rangle\!\rangle \backslash x \quad E \vdash A \ type}{E \vdash \langle\!\langle R \mid x{:}A \rangle\!\rangle <: \langle\!\langle\rangle\!\rangle}$$

(S2)
$$\frac{E \vdash R <: \langle\!\langle\rangle\!\rangle}{E \vdash R \backslash x <: \langle\!\langle\rangle\!\rangle}$$

Case (2). A subtype of an extended record type will be another extended record type, provided all respective components are in the subtype relation:

(S3)
$$\frac{E \vdash R <: S <: \langle\!\langle\rangle\!\rangle \backslash x \quad E \vdash A <: B}{E \vdash \langle\!\langle R \mid x{:}A \rangle\!\rangle <: \langle\!\langle S \mid x{:}B \rangle\!\rangle}$$

The condition $A<:B$ says that we can produce a subtype by weakening the type of a given field. The condition $R<:S$ tells us that we can produce a subtype either (a) by weakening other fields inductively, because of (S3) itself, or (b) by requiring the presence of additional components, because of (S1), or (c) by requiring the absence of additional components, for example y, because from (S2) we are able to derive $\langle\!\langle\rangle\!\rangle \backslash yx <: \langle\!\langle\rangle\!\rangle \backslash x$.

Case (3). The subtype rule for restricted types is semantically straightforward: if every r in R occurs in S, then every $r\backslash x$ in $R\backslash x$ occurs in $S\backslash x$:

(S4)
$$\frac{E \vdash R <: S <: \langle\!\langle\rangle\!\rangle}{E \vdash R \backslash x <: S \backslash x}$$

Case (4). We have to consider the subtypes of record type extractions; that is situations of the form $R.x <: T.x$, or more generally $R.x <: A$ under an assumption $R <: \langle\!\langle S \mid x{:}B \rangle\!\rangle$. If R can be converted to the form $R=\langle\!\langle R' \mid x{:}A \rangle\!\rangle$, then the extraction $R.x$ simplifies and no special rule is required to deduce $R.x<:A$. But if R is a type variable, for example, the following rule is necessary:

(S5)
$$\frac{E \vdash R <: \langle\!\langle S \mid x{:}A \rangle\!\rangle <: \langle\!\langle\rangle\!\rangle}{E \vdash R.x <: A}$$

This says that if R has an x field of type A, then $R.x$ is a subtype of A (and possibly equal to A).

Finally, there is a another subtyping rule that we must consider. If every record r in R has an x field, then any such r is described also by the type $\langle\!\langle R\backslash x \mid x{:}R.x\rangle\!\rangle$, since $r\backslash x$ is described by $R\backslash x$ and the x field of r is described by $R.x$. Therefore we have the following inclusion:

$$\text{(S6)}\quad \frac{E \vdash R <: \langle\!\langle S \mid x{:}A\rangle\!\rangle <: \langle\!\langle\rangle\!\rangle}{E \vdash R <: \langle\!\langle R\backslash x \mid x{:}R.x\rangle\!\rangle}$$

The inverse inclusion is not necessarily valid, although it might seem natural to require it as we shall see later.

The rule (S6) can be used in the following derivation, which provides a "symmetrical" version of (S5) as a derived rule:

$$\begin{array}{ll} & \dfrac{E \vdash R <: S <: \langle\!\langle T \mid x{:}A\rangle\!\rangle <: \langle\!\langle\rangle\!\rangle}{} \\ \text{(S6)} & \dfrac{E \vdash S <: \langle\!\langle S\backslash x \mid x{:}S.x\rangle\!\rangle}{} \\ \text{(G2)} & \dfrac{E \vdash R <: \langle\!\langle S\backslash x \mid x{:}S.x\rangle\!\rangle}{} \\ \text{(S5)} & E \vdash R.x <: S.x \end{array}$$

In absence of (S6), the derived rule above would have to be taken as primitive, replacing (S5).

3.6 Record typing and equivalence

Now that we have seen the rules for type equivalence and subtyping, the rules for record values follow rather naturally. The only subtle point is about the empty record. We must be able to assign it a type which lacks any given set of labels. This is obtained by repeatedly applying the following two rules:

$$\text{(I1)}\quad \frac{\vdash E\ env}{E \vdash \langle\rangle : \langle\!\langle\rangle\!\rangle} \qquad \text{(I2)}\quad \frac{E \vdash \langle\rangle : R <: \langle\!\langle\rangle\!\rangle}{E \vdash \langle\rangle : R\backslash x}$$

The remaining constructions on record values are typed by the corresponding constructions on record types, given the appropriate assumptions:

$$\text{(I3)}\quad \frac{E \vdash r{:}R <: \langle\!\langle\rangle\!\rangle\backslash x \quad E \vdash a{:}A}{E \vdash \langle r \mid x{=}a\rangle : \langle\!\langle R \mid x{:}A\rangle\!\rangle} \qquad \text{(E1)}\quad \frac{E \vdash r{:}R <: \langle\!\langle\rangle\!\rangle}{E \vdash r\backslash x : R\backslash x} \qquad \text{(E2)}\quad \frac{E \vdash r{:}\langle\!\langle R \mid x{:}A\rangle\!\rangle <: \langle\!\langle\rangle\!\rangle}{E \vdash r.x : A}$$

As we did in the previous section, we can use the rule (S6) to derive a "symmetrical" version of (I2):

$$\dfrac{\dfrac{\dfrac{E \vdash r{:}R <: \langle\!\langle S \mid x{:}A \rangle\!\rangle <: \langle\!\langle \rangle\!\rangle}{E \vdash R <: \langle\!\langle R \setminus x \mid x{:}R.x \rangle\!\rangle}\ (S6)}{E \vdash r{:}\langle\!\langle R \setminus x \mid x{:}R.x \rangle\!\rangle}\ (G3)}{E \vdash r.x : R.x}\ (E2)$$

Finally, we have to examine the rules for record value equivalence. These rules are formally very similar to the ones already discussed for record type equivalence; record extensions can be permuted, record components can be extracted, and restrictions can be permuted and pushed inside extensions, sometimes cancelling each other.

The main formal difference between these and the rules for types is that we equate $\langle\rangle \setminus x \leftrightarrow \langle\rangle$. Hence, restriction can always be completely eliminated from variable-free records.

Because of the formal similarity we omit a detailed discussion; the complete set of rules for our type system follows in the next section.

3.7 Type rules

We can now summarize and complete the rules for record types and values, along with selected auxiliary rules. These rules are designed to be immersed in a second-order λ-calculus with bounded quantification (see [Cardelli Wegner 85]), and possibly with recursive values and types.

We only list the names of the rules that have already been discussed.

Environments

(ENV1)...(ENV4), (VAR1)...(VAR4)

General properties of <: and ↔

(G1)...(G3)

(G4)
$$\dfrac{E \vdash A \leftrightarrow B}{E \vdash B \leftrightarrow A}$$

(G5)
$$\dfrac{E \vdash A \leftrightarrow B \quad E \vdash B \leftrightarrow C}{E \vdash A \leftrightarrow C}$$

(G6)
$$\dfrac{E \vdash a \leftrightarrow b : A}{E \vdash b \leftrightarrow a : A}$$

(G7)
$$\dfrac{E \vdash a \leftrightarrow b : A \quad E \vdash b \leftrightarrow c : A}{E \vdash a \leftrightarrow c : A}$$

Formation

(F1)...(F4)

Subtyping

(S1)...(S6)

Introduction/Elimination

(I1)...(I3), (E1), (E2)

Type Congruence

(TC1)
$$\frac{\vdash E\ env}{E \vdash \langle\!\rangle \leftrightarrow \langle\!\rangle}$$

(TC2)
$$\frac{E \vdash X\ type}{E \vdash X \leftrightarrow X}$$

(TC3)
$$\frac{E \vdash R \leftrightarrow S <: \langle\!\rangle \backslash x \quad E \vdash A \leftrightarrow B}{E \vdash \langle\!\langle R \mid x{:}A \rangle\!\rangle \leftrightarrow \langle\!\langle S \mid x{:}B \rangle\!\rangle}$$

(TC4)
$$\frac{E \vdash R \leftrightarrow S <: \langle\!\rangle}{E \vdash R\backslash x \leftrightarrow S\backslash x}$$

(TC5)
$$\frac{E \vdash R \leftrightarrow S <: \langle\!\langle T \mid x{:}A \rangle\!\rangle <: \langle\!\rangle}{E \vdash R.x \leftrightarrow S.x}$$

Type Equivalence

(TE1)...(TE8)

Value Congruence

(VC1a)
$$\frac{\vdash E\ env}{E \vdash \langle\rangle \leftrightarrow \langle\rangle : \langle\!\rangle}$$

(VC2)
$$\frac{E \vdash x : A}{E \vdash x \leftrightarrow x : A}$$

(VC3)
$$\frac{E \vdash r \leftrightarrow s : R <: \langle\!\rangle \backslash x \quad E \vdash a \leftrightarrow b : A}{E \vdash \langle r \mid x{=}a \rangle \leftrightarrow \langle s \mid x{=}b \rangle : \langle\!\langle R \mid x{:}A \rangle\!\rangle}$$

(VC4)
$$\frac{E \vdash r \leftrightarrow s : R <: \langle\!\rangle}{E \vdash r\backslash x \leftrightarrow s\backslash x : R\backslash x}$$

(VC5)
$$\frac{E \vdash r \leftrightarrow s : R <: \langle\!\langle S \mid x{:}A \rangle\!\rangle <: \langle\!\rangle}{E \vdash r.x \leftrightarrow s.x : R.x}$$

Value Equivalence

(VE1)
$$\frac{E \vdash r{:}R <: \langle\!\rangle \backslash xy \quad E \vdash a{:}A \quad E \vdash b{:}B \quad x \neq y}{E \vdash \langle \langle r \mid x{=}a \rangle \mid y{=}b \rangle \leftrightarrow \langle \langle r \mid y{=}b \rangle \mid x{=}a \rangle : \langle\!\langle \langle\!\langle R \mid x{:}A \rangle\!\rangle \mid y{:}B \rangle\!\rangle}$$

(VE2)
$$\frac{\vdash E\ env}{E \vdash \langle\rangle \backslash x \leftrightarrow \langle\rangle : \langle\!\rangle}$$

(VE3)
$$\frac{E \vdash r{:}R <: \langle\!\rangle \backslash x}{E \vdash r\backslash x \leftrightarrow r : R}$$

(VE4)
$$\frac{E \vdash r{:}R <: \langle\!\rangle}{E \vdash r\backslash xy \leftrightarrow r\backslash yx : R\backslash xy}$$

(VE5)
$$\frac{E \vdash r{:}\langle\!\langle R \mid x{:}A \rangle\!\rangle <: \langle\!\rangle \quad x \neq y}{E \vdash r\backslash y.x \leftrightarrow r.x : A}$$

(VE6)
$$\frac{E \vdash r{:}R{<:}\langle\!\langle\rangle\!\rangle\backslash x \quad E \vdash a{:}A}{E \vdash \langle r \mid x{=}a\rangle\backslash x \leftrightarrow r : R}$$

(VE7)
$$\frac{E \vdash r{:}R{<:}\langle\!\langle\rangle\!\rangle\backslash x \quad E \vdash a{:}A \quad x{\neq}y}{E \vdash \langle r \mid x{=}a\rangle\backslash y \leftrightarrow \langle r\backslash y \mid x{=}a\rangle : \langle\!\langle R \mid x{:}A\rangle\!\rangle\backslash y}$$

(VE8)
$$\frac{E \vdash r{:}R{<:}\langle\!\langle\rangle\!\rangle\backslash x \quad E \vdash a{:}A}{E \vdash \langle r \mid x{=}a\rangle.x \leftrightarrow a : A}$$

(VE9)
$$\frac{E \vdash r{:}\langle\!\langle R \mid y{:}B\rangle\!\rangle\backslash x{<:}\langle\!\langle\rangle\!\rangle \quad E \vdash a{:}A \quad x{\neq}y}{E \vdash \langle r \mid x{=}a\rangle.y \leftrightarrow r.y : B}$$

(VE10)
$$\frac{E \vdash r{:}R{<:}\langle\!\langle S \mid x{:}A\rangle\!\rangle{<:}\langle\!\langle\rangle\!\rangle}{E \vdash r \leftrightarrow \langle r\backslash x \mid x{=}r.x\rangle : R}$$

Special rules

In the following sections we discuss the rules (VC1b) and (TE9) below; these are valid only with respect to particular semantic interpretations.

(VC1b)
$$\frac{E \vdash r{:}\langle\!\langle\rangle\!\rangle \quad E \vdash s{:}\langle\!\langle\rangle\!\rangle}{E \vdash r \leftrightarrow s : \langle\!\langle\rangle\!\rangle}$$

(TE9)
$$\frac{E \vdash R{<:}\langle\!\langle S \mid x{:}A\rangle\!\rangle{<:}\langle\!\langle\rangle\!\rangle}{E \vdash R \leftrightarrow \langle\!\langle R\backslash x \mid x{:}R.x\rangle\!\rangle}$$

In presence of (TE9), the rule (S6) is redundant, and the rules (TC5) and (VC5) are implied by the simpler (TC5b) and (VC5b) below.

(TC5b)
$$\frac{E \vdash R \leftrightarrow \langle\!\langle S \mid x{:}A\rangle\!\rangle{<:}\langle\!\langle\rangle\!\rangle}{E \vdash R.x \leftrightarrow A}$$

(VC5b)
$$\frac{E \vdash r \leftrightarrow s : \langle\!\langle R \mid x{:}A\rangle\!\rangle{<:}\langle\!\langle\rangle\!\rangle}{E \vdash r.x \leftrightarrow s.x : A}$$

Properties

Lemma 3.7.1:
(1) If $E \vdash A$ *type*, then $\vdash E$ *env*.
(2) If $E \vdash A <: B$, then $\vdash E$ *env*.

Lemma 3.7.2:
(1) If $E \vdash A \leftrightarrow B$, then $E \vdash A$ *type* and $E \vdash B$ *type*.
(2) If $E \vdash A <: B$, then $E \vdash A$ *type* and $E \vdash B$ *type*.

3.8 Semantics of the pure calculus of records

Our stated intent is to define a second-order type system for record structures. However, models of such a system are rather complex, and outside the scope of this paper.

In this section we provide a simple set-theoretical model of the pure calculus of records, without any additional functional or polymorphic structure. The intent here is to show the plausibility of the inference rules for records, by proving their soundness with respect to a natural model.

This model is natural because it embodies the strong set-theoretical intuitions of subtyping seen as a subset relation, and of records seen as finite tuples. Although this model does not extend to more complex language features, it exhibits the kind of simple-minded but (usually) sound reasoning that guides the design and implementation of object-oriented languages.

Syntax

We start with the language implied by the type rules of section 3.7. Since no basic non-record values are expressible in this calculus, we must make some arbitrary choices to get started. To this end, we will consider an extension of the pure calculus with any collection G_1 , G_2 , ... of basic (ground) type symbols and an arbitrary collection of subtype relations $G_i <: G_j$ between them. To incorporate these new symbols into the calculus, we add the following two rules (which preserve lemmas 3.7.1 and 3.7.2):

$$\frac{\vdash E\ env}{E \vdash G_i\ type} \qquad \frac{\vdash E\ env}{E \vdash G_i <: G_j} \quad \text{(as appropriate)}$$

For simplicity we do not introduce value constants, instead we work with environments containing assumptions of the form $k : G_i$.

We will now construct a model of the extended calculus.

Semantic domains

In the following, we rely largely on context to distinguish between syntactic expressions and semantic expressions, and we often identify terms with their denotations.

We start by choosing some fixed set of labels L, and a collection of sets $\mathcal{G}_1$, $\mathcal{G}_2$, ... corresponding to the type symbols G_1 , G_2 , ... such that $\mathcal{G}_i \subseteq \mathcal{G}_j$ if $G_i <: G_j$ is a subtyping axiom.

For simplicity, we assume that no element of any $\mathcal{G}_i$ is a finite partial function on L (i.e. a record, as we shall see shortly). This assumption is useful when we define the subtype relations of section 3.9.

Since $\langle\!\langle\rangle\!\rangle$ serves as a type of all records, we will need some value space closed under record formation. This property may be accomplished by regarding records as finite functions from L to values, and using *ranked* values with rank $< \omega$. We use $A \rightharpoonup_{\text{fin}} B$ for the set of partial functions from A to B with finite domain, $f(x)\uparrow$ to indicate that the partial function f is undefined at x, and $f(x)\downarrow$ to indicate that f is defined at x.

Define set $\mathcal{R}_i$ of records of rank i, and set $\mathcal{V}_i$ of values of rank i, as follows:

$$\begin{array}{lll} \mathcal{V}_0 = \bigcup_j \mathcal{G}_j & & \mathcal{V}_{i+1} = \mathcal{R}_i \cup \mathcal{V}_i \\ \mathcal{R}_0 = L \rightarrow_{\text{fin}} \mathcal{V}_0 & & \mathcal{R}_{i+1} = L \rightarrow_{\text{fin}} \mathcal{V}_{i+1} \end{array}$$

$$\begin{array}{lll} \mathcal{R} = \bigcup_{i<\omega} \mathcal{R}_i & & \text{the set of } \textit{records} \\ \mathcal{V} = \bigcup_{i<\omega} \mathcal{V}_i & & \text{the set of } \textit{values} \end{array}$$

The essential properties of this construction are summarized by the relationship:

$$\mathcal{R} = (L \rightarrow_{\text{fin}} \mathcal{V}) \subseteq \mathcal{V}$$

It is clear by construction that $\mathcal{R}_i \subseteq \mathcal{V}_{i+1}$ and so $\mathcal{R} \subseteq \mathcal{V}$. To see that $\mathcal{R} = L \rightarrow_{\text{fin}} \mathcal{V}$, we first show that $L \rightarrow_{\text{fin}} \mathcal{V} \subseteq \mathcal{R}$. If $r \in L \rightarrow_{\text{fin}} \mathcal{V}$, then since $dom(r)$ is finite there is some i with $range(r) \subseteq \mathcal{V}_i$; hence $r \in \mathcal{R}_i \subseteq \mathcal{R}$. The converse follows from the fact that if $r \in \mathcal{R}$, then $r \in \mathcal{R}_i = (L \rightarrow_{\text{fin}} \mathcal{V}_i) \subseteq L \rightarrow_{\text{fin}} \mathcal{V}$.

We now summarize the notation used to describe the semantic interpretation of syntactic constants and operators:

$\varnothing$	$=$	$\lambda y \in L.\ \uparrow$
$r\text{-}x$	$=_{\text{def}}$	$\lambda y \in L.$ *if* $y=x$ *then* $\uparrow$ *else* $r(y)$ provided $r \in \mathcal{R}$ and $x \in L$
$r[x=a]$	$=_{\text{def}}$	$\lambda y \in L.$ *if* $y=x$ *then* a *else* $r(y)$ provided $r \in \mathcal{R}$, $x \in L$, $a \in \mathcal{V}$, and $x \notin dom(r)$.
$r(x)$		is well-defined, provided $r \in \mathcal{R}$, $x \in L$, and $x \in dom(r)$.

Lemma 3.8.1:

(1) The empty record $\varnothing$ is an element of $\mathcal{R}$.
(2) For any $r \in \mathcal{R}$ we have $r\text{-}x \in \mathcal{R}$.
(3) If $r \in \mathcal{R}$ is not defined on x, then for any $a \in \mathcal{V}$ we have $r[x=a] \in \mathcal{R}$.
(4) If $r \in \mathcal{R}$ is defined on x, then $r(x) \in \mathcal{V}$.

Types and type operations

Types are interpreted as subsets of our global value set; hence we have a type of all values, and a type of all records. Subtyping is interpreted as set inclusion.

We introduce the following notation for operations on record types:

$R\text{-}x \quad =_{\text{def}} \quad \{r\text{-}x \mid r \in R\}$
if $R \subseteq \mathcal{R}$

$R[x{:}A] \quad =_{\text{def}} \quad \{r[x{=}a] \mid r \in R, a \in A\}$
if $R \subseteq \mathcal{R}\text{-}x$ (R undefined on x) and $A \subseteq \mathcal{V}$

$R(x) \quad =_{\text{def}} \quad \{r(x) \mid r \in R\}$
if $R \subseteq S[x{:}A]$ for some $S \subseteq \mathcal{R}$ and $A \subseteq \mathcal{V}$

Lemma 3.8.2:
Under the conditions stated above, the sets $R\text{-}x$ and $R[x{:}A]$ are subsets of $\mathcal{R}$, and the sets $R(x)$ are subsets of $\mathcal{V}$.

Interpretation of judgments

An *assignment* ρ is a partial map from type variables to subsets of $\mathcal{V}$, and from ordinary variables to elements of $\mathcal{V}$. We say that an assignment ρ *satisfies* an environment E if the following conditions are satisfied:

If X in E, then $\rho(X) \subseteq \mathcal{V}$
If $X <: A$ in E, then $\rho(X) \subseteq A_\rho \subseteq \mathcal{V}$
If $x : A$ in E, then $\rho(x) \in A_\rho \subseteq \mathcal{V}$

where A_ρ is the type defined by A under the assignment ρ. Similarly, by a_ρ we indicate the value of a term a under an assignment ρ for its free variables.

The judgments of our system are interpreted as follows.

$\vdash E\ env \quad \approx$ for every initial segment $E', X<:A$ or $E', x:A$ of E, if ρ satisfies E' then $A_\rho \subseteq \mathcal{V}$.
$E \vdash A\ type \quad \approx$ $A_\rho \subseteq \mathcal{V}$, for every ρ satisfying E.
$E \vdash A <: B \quad \approx$ $A_\rho \subseteq B_\rho \subseteq \mathcal{V}$, for every ρ satisfying E.
$E \vdash A \leftrightarrow B \quad \approx$ $A_\rho = B_\rho \subseteq \mathcal{V}$, for every ρ satisfying E.
$E \vdash a : A \quad \approx$ $a_\rho \in A_\rho \subseteq \mathcal{V}$, for every ρ satisfying E.
$E \vdash a \leftrightarrow b : A \quad \approx$ $a_\rho = b_\rho \in A_\rho \subseteq \mathcal{V}$, for every ρ satisfying E.

Type and value expressions are interpreted using:

$\langle\!\langle\rangle\!\rangle \quad \approx \quad \mathcal{R}$
$R \setminus x \quad \approx \quad R\text{-}x$
$\langle\!\langle R \mid x{:}A \rangle\!\rangle \quad \approx \quad R[x{:}A]$
$R.x \quad \approx \quad R(x)$

$$
\begin{array}{lcl}
\langle\rangle & \approx & \varnothing \\
r\backslash x & \approx & r\text{-}x \\
\langle r \mid x{=}a\rangle & \approx & r[x{=}a] \\
r.x & \approx & r(x)
\end{array}
$$

Soundness

Finally, we can show that this semantics satisfies the type rules. More precisely, we consider the system *S1* consisting of all the rules listed in section 3.7, except for the special rules (VC1b) and (TE9).

Theorem 3.8.3 (soundness):
The inference rules of system *S1* are sound with respect to the interpretation of judgments given in this section.

3.9 Other semantic constructions

The type equivalence rule below seems very natural semantically. It also simplifies the types associated with the override operation, and has application to extensional models studied in the next section.

(TE9)
$$\frac{E \vdash R <: \langle\!\langle S \mid x{:}A\rangle\!\rangle <: \langle\!\langle\rangle\!\rangle}{E \vdash R \leftrightarrow \langle\!\langle R\backslash x \mid x{:}R.x\rangle\!\rangle}$$

In the simple model described in section 3.8, it is easy to see that if $R \subseteq \langle\!\langle x{:}A\rangle\!\rangle$, then, as required by (S6):

$$R \subseteq \langle\!\langle R\backslash x \mid x{:}R.x\rangle\!\rangle$$

The reason is that every record r in R has an x component $r(x) \in R(x)$, and remaining components $r\text{-}x$ in $R\text{-}x$. However, it is not necessarily true that every combination of $r\text{-}x$ from $R\text{-}x$ and $r(x)$ from $R(x)$ occur together in a single record in R. For example, the set of records:

$$R = \{\langle x{=}1, y{=}true\rangle, \langle x{=}0, y{=}false\rangle\}$$

is clearly a subset of $\langle\!\langle x{:}Int\rangle\!\rangle$. However, $R \neq \langle\!\langle R\backslash x \mid x{:}R.x\rangle\!\rangle$ since the records $\langle x{=}1, y{=}false\rangle$ and $\langle x{=}0, y{=}true\rangle$ do not appear in R. In category-theoretic terms, the equation $R = \langle\!\langle R\backslash x \mid x{:}R.x\rangle\!\rangle$ says that R is the product of $R\backslash x$ and $R.x$.

In the full paper we present a variant of the construction of section 3.8 in which rule (TE9) is sound. Since we are ultimately interested in polymorphism

and bounded quantification, we construct a model with $R = \langle\!\langle R\backslash x \mid x{:}R.x \rangle\!\rangle$ for every semantic type R with $R.x$ defined. The construction uses the same collection of values as before, but allows only certain subsets of $\mathcal{V}$ as types. In this way we eliminate sets of records which violate (TE9).

Another construction arises from the following inference rule, which gives us an extensional equality between records:

(VC1b)
$$\frac{E \vdash r{:}\langle\!\langle\rangle\!\rangle \quad E \vdash s{:}\langle\!\langle\rangle\!\rangle}{E \vdash r \leftrightarrow s : \langle\!\langle\rangle\!\rangle}$$

The intuitive explanation of this rule is that if r and s both belong to $\langle\!\langle\rangle\!\rangle$, then r and s are indistinguishable. In fact, assume r and s differ at some label x. We cannot use $r.x$ or $s.x$ to distinguish them since neither is well-typed; if we use $r\backslash x$ or $s\backslash x$ then we simply remove the difference.

In addition to giving us more equations between records of type $\langle\!\langle\rangle\!\rangle$, rule (VC1b) implies the following extensionality property: for any $r,s : \langle\!\langle x_1{:}A_1 , \ldots , x_k{:}A_k \rangle\!\rangle$, we have $r \leftrightarrow s : \langle\!\langle x_1{:}A_1 , \ldots , x_k{:}A_k \rangle\!\rangle$ iff $r.x_i \leftrightarrow s.x_i : A_i$ for $i = 1 \ldots k$. The straightforward proof of this uses $r\backslash x_1 \ldots x_k \leftrightarrow s\backslash x_1 \ldots x_k : \langle\!\langle\rangle\!\rangle$ and the value congruence rules.

In the full paper, we construct a model of the pure record calculus satisfying (TE9) and (VC1b). In this construction, (TE9) is essential; we do not know how to construct an extensional model satisfying (VC1b) without requiring that record types satisfy $R = \langle\!\langle R\backslash x \mid x{:}R.x \rangle\!\rangle$. The main use of (TE9) lies in showing that if R is a record type with extensional equality, then both $R\text{-}x$ and $R(x)$, when defined, are extensional record types.

3.10 Normalization and decidability

Even though the basic ideas behind the record calculus are relatively simple, the formal system has quite a few rules. As a consequence, it is not easy to see, by inspection, how we could determine whether a supposed type A is well-formed, or whether a record expression has type R.

In this section, we outline a proof that all of the basic properties of the calculus are decidable, using relatively natural algorithms. In the process, we show that every type expression has a unique normal form (modulo permuting the order of fields) and every typable record expression has a *principal type* in each suitable environment.

The first properties we consider are deciding whether a supposed environment E is well-formed and whether a given A is a well-formed type expression in E. A quick glance at the formation rules shows that in order to determine whether a type is well-formed we must be able to decide the

following apparently simple properties; assuming $E \vdash R$ *type* is derivable, we want to know whether $E \vdash R <: \langle\!\langle\rangle\!\rangle \backslash x$ and whether there exist S and A such that $E \vdash R <: \langle\!\langle S \mid x{:}A \rangle\!\rangle$. Therefore, we consider these first. Once we develop a simple method for these, it is easy to check whether a type or environment is well-formed.

For each derivable $E \vdash R$ *type*, we define a labeled tree *Tree*($E \vdash R$ *type*) with:

edges:	labeled by field names
vertices:	labeled by finite sets of field names

If v is a vertex in *Tree*($E \vdash R$ *type*), we call the finite set of field names at v the *absent set at v*.

Intuitively, if $p = x_1 x_2 \ldots x_k$ is a path from the root of *Tree*($E \vdash R$ *type*) and $N = \{y_1, y_2, \ldots, y_l\}$ is the absent set of the vertex designated by this path, then:

$$E \vdash (..(R.x_1).x_2 \ldots).x_k \textit{ type}$$
$$E \vdash (..(R.x_1).x_2 \ldots).x_k <: \langle\!\langle\rangle\!\rangle \backslash y_1 y_2 \ldots y_l$$

A convenient notational shorthand is to write $R.p$ for $(..(R.x_1).x_2 \ldots).x_k$, where p is the path $p = x_1 x_2 \ldots x_k$. If $p = \varepsilon$ is the empty path, then we may write $R.\varepsilon$ for R. If e is an edge leading from the root of a tree to the root of some subtree, we call e a *root edge*.

The inductive definition of *Tree*($E \vdash R$ *type*) is given in the full paper.

Lemma 3.10.1:

Suppose $E \vdash R$ *type* and let $T =$ *Tree*($E \vdash R$ *type*).
(1) If p is a path in T, then $E \vdash R.p$ *type*.
(2) If x is in the absent set of T at position p, then $E \vdash R.p <: \langle\!\langle\rangle\!\rangle \backslash x$.

Lemma 3.10.2:

Suppose $E \vdash R$ *type* and let $T =$ *Tree*($E \vdash R$ *type*).
There is a semantic model $\mathcal{M}$ and assignment ρ such that:
(1) If p is a sequence of labels which is not a path in T, then there is some record r in R_ρ with $r.p$ undefined.
(2) If p is a path in T with x absent from every record in $(R.p)_\rho$, then x is in the absent set of T at the vertex located at p.

By constructing trees of absent sets, it is relatively easy to decide whether a purported environment or type expression is well-formed. The basic idea is simply to check whether $\vdash E$ *env* or $E \vdash R$ *type* by reading the environment and formation rules backwards. This gives us mutually recursive procedures which rely on *Tree*($E \vdash R$ *type*) in checking the hypotheses of (F2) and (F4).

Theorem 3.10.3:

Given environment E and expression A, there are mutually recursive procedures which decide whether $\vdash E$ *env* and $E \vdash A$ *type*.

The next problems to consider are, given well-formed types $E \vdash A$ *type* and $E \vdash B$ *type*, whether $E \vdash A \leftrightarrow B$ or $E \vdash A <: B$. Since type equality may be used to prove subtyping assertions, both depend on our choice of type equality rules. For definiteness, let us assume we have (TE9). Similar results seem to hold without (TE9), but we have not checked the details.

Theorem 3.10.4:

Given $E \vdash A$ *type* and $E \vdash B$ *type*, there are straightforward algorithms to determine whether $E \vdash A \leftrightarrow B$ or $E \vdash A <: B$. Moreover, the proof rules are semantically complete for deducing type equality and subtype assertions.

The final algorithmic problem is, given $E \vdash R$ *type* and an expression r, determine whether $E \vdash r:R$.

Since we can decide whether one type is a subtype of another, it suffices to compute a minimal type S with $E \vdash r:S$ and check whether $E \vdash S <: R$.

However, most record expressions do not have a minimal type. This stems from the fact that for any sequence $x_1 \ldots x_k$ of labels, we have $\langle\rangle : \langle\langle\rangle\rangle \backslash x_1 \ldots x_k$, and we can always obtain a smaller type by adding more labels. To get around this problem, we use *type schemas* that contain sequence variables. We show that each typable record expression r has a scheme S such that every type for r is a supertype of some instance of S. This allows us to test whether a record expression has any given type.

Theorem 3.10.5:

There is an algorithm PTS(E,r) such that, given $\vdash E$ *env* and an expression r, if $E \vdash r:R$ then *PTS*(E,r) succeeds, producing S with $E \vdash S' <: R$ for some instance S' of S. Otherwise, *PTS*(E,r) fails. Furthermore, given $S = PTS(E,r)$ and $E \vdash R$ *type*, it is easy to compute the smallest instance S' of S such that if any instance is a subtype of R, then $E \vdash S' <: R$.

This concludes our investigation of decidability properties. We leave extensions of these properties to functions and polymorphism for further work.

4. Conclusions

We have investigated a theory of record operations in presence of type variables and subtyping. The intent is to embed this record calculus in a polymorphic λ-calculus, thus providing a full second-order theory of record structures and their types. Although we have not investigated the type inference problem for this calculus, we have provided typechecking and subtyping algorithms. We have also presented several models of the basic record calculus; a full second-order model is left for future work.

The result is a very flexible system for typing programs that manipulate records. In particular, polymorphism and subtyping are incorporated in full generality. We expect that this theory will be useful in analyzing fundamental aspects of object-oriented programming.

Acknowledgements

We would like to acknowledge G. Longo and E. Moggi, for several clarifying discussions.

References

[Breazu-Tannen Coquand Gunter Scedrov 89] V.Breazu-Tannen, T.Coquand, C.Gunter, A.Scedrov: ***Inheritance and explicit coercion***, Proc. of the Fourth Annual Symposium on Logic in Computer Science, 1989.

[Bruce Longo 88] K.B.Bruce, G.Longo: ***Modest models for inheritance and explicit polymorphism***, Proc. of the Third Annual Symposium on Logic in Computer Science, 1988.

[Bruce Meyer Mitchell 89] K.B.Bruce, A.R.Meyer, J,C.Mitchell: ***The semantics of second order lambda calculus***, Information and Computation, 1989 (to appear).

[Cardelli Donahue Glassman Jordan Kalsow Nelson 88] L.Cardelli, J.Donahue, L.Glassman, M.Jordan, B.Kalsow, G.Nelson: ***Modula-3 report***, Research Report n.31, DEC Systems Research Center, Sep. 1988.

[Cardelli 84&88] L.Cardelli: ***A semantics of multiple inheritance***, in Information and Computation 76, pp 138-164, 1988. (First appeared in Semantics of Data Types, G.Kahn, D.B.MacQueen and G.Plotkin Ed. Lecture Notes in Computer Science n.173, Springer-Verlag 1984.)

[Cardelli Wegner 85] L.Cardelli, P.Wegner: ***On understanding types, data abstraction and polymorphism***, Computing Surveys, Vol 17 n. 4, pp 471-522, December 1985.

[Curien Ghelli 89] P.-L.Curien, G.Ghelli: ***Coherence of subsumption***, to appear.

[Dahl Nygaard 66] O.Dahl, K.Nygaard: ***Simula, an Algol-based simulation language***, Communications of the ACM, Vol 9, pp. 671-678, 1966.

[Girard 71] J-Y.Girard: ***Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types***, Proceedings of the second Scandinavian logic symposium, J.E.Fenstad Ed. pp. 63-92, North-Holland, 1971.

[Girard 72] J-Y.Girard: ***Interprétation fonctionelle et élimination des coupures dans l'arithmétique d'ordre supérieur***, Thèse de doctorat d'état, University of Paris, 1972.

[Jategaonkar Mitchell 88] L.A.Jategaonkar, J.C.Mitchell: ***ML with extended pattern matching and subtypes***, Proc. of the ACM Conference on Lisp and Functional Programming, pp.198-211, 1988.

[Longo Moggi 88] G.Longo, E.Moggi: ***Constructive natural deduction and its 'ω-set' interpretation***, Report CMU-CS-88-131, CMU, Dept. of Computer Science, 1988.

[Meyer 88] B.Meyer: ***Object-oriented software construction***, Prentice Hall, 1988.

[Milner 78] R.Milner: ***A theory of type polymorphism in programming***, Journal of Computer and System Science 17, pp. 348-375, 1978.

[Mitchell 84] J.C.Mitchell: ***Coercion and type inference***, Proc. of the 11th ACM Symposium on Principles of Programming Languages, pp.175-185, 1984.

[Mitchell 86] J.C.Mitchell: ***A type inference approach to reduction properties and semantics of polymorphic expressions***, Proc. Symposium on Lisp and Functional Programming, pp.308-319, 1986. (Revised version to appear in Logic Foundations of Functional Programming, ed. G. Huet, Addison-Wesley, 1989.)

[Mitchell 90] J.C.Mitchell: ***Type systems for programming languages***, in Handbook of Theoretical Computer Science, ed. J. van Leeuwen et al. North Holland, 1990 (to appear).

[Ohori Buneman Breazu-Tannen 88] A.Ohori, P.Buneman, V.Breazu-Tannen: ***Database programming in Machiavelli - a polymorphic languaage with static type inference***, Report MS-CIS-88-103, University of Pennsylvania, Computer and Information Science Dept., 1988.

[Ohori Buneman 88] A.Ohori, P.Buneman: ***Type inference in a database programming language***, Proc. of the ACM Conference on LISP and Functional Programming, pp.174-183, Snowbird, Utah, 1988.

[Rémy 89] D. Rémy: ***Typechecking records and variants in a natural extension of ML***, Proc. of the 16th ACM Symposium on Principles of Programming Languages, pp.77-88, 1989.

[Reynolds 74] J.C.Reynolds: ***Towards a theory of type structure***, in Colloquium sur la programmation pp. 408-423, Springer-Verlag Lecture Notes in Computer Science, n.19, 1974.

[Schaffert Cooper Bullis Kilian Wilpolt 86] C.Schaffert, T.Cooper, B.Bullis, M.Kilian, C.Wilpolt: ***An introduction to Trellis/Owl***, Proc. OOPSLA'86.

[Stroustrup 86] B.Stroustrup: ***The C++ programming language***, Addison-Wesley 1986.

[Wand 87] M.Wand: ***Complete Type Inference for Simple Objects***, Proc. of the Second Annual Symposium on Logic in Computer Science, June 1987, Cornell University.

[Wand 89] M.Wand: ***Type inference for record concatenation and multiple inheritance***, Proc. of the Fourth Annual Symposium on Logic in Computer Science, pp. 92-97, 1989.