Operations on Records

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.

Appears in: Theoretical Aspects of Object-Oriente Programming, C.Gunter, J.C.Mitchell Eds. MIT Press, 1994.

Appears in: Mathematical Structures in Computer Science, vol 1, pp. 3-48, 1991

SRC Research Report 48, August 25, 1989. Revised October 22, 1993.

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 1988] the basic notions of record types, as intended here, were defined in the context of a first-order type system for fixed-size records. Then Wand [Wand 1987] 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 1988] [Rémy 1989], and again in [Wand 1989]. The resulting system provides a flexible integration of record types and Milner-style type inference [Milner 1978].

Meanwhile [Cardelli, Wegner 1985] defined a full second-order extension of the system with fixed-size records, based on techniques from [Mitchell 1984]. 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. We give a normalization procedure for record types, and we show soundness of the rules with respect to a simple semantics for the pure calculus of records. Finally, we discuss applications and extensions of the basic calculus.

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* $\langle r|x{=}a\rangle$; 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 $\langle r|x{=}a|y{=}b\rangle$ for $\langle\langle r|x{=}a\rangle|y{=}b\rangle$.
- *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 1987], 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 1987] and [Jategaonkar, Mitchell 1988], 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 1989]). 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.:

$$\begin{array}{lll} \langle x{=}3\rangle & =_{\text{def}} & \langle\langle\rangle \mid x{=}3\rangle \\ \langle x{=}3, y{=}true\rangle & =_{\text{def}} & \langle\langle\langle\rangle \mid x{=}3\rangle \mid y{=}true\rangle \end{array}$$

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:

$$\langle\langle\rangle \mid x{=}3 \mid y{=}true\rangle \quad\leftrightarrow\quad \langle\langle\langle\rangle\backslash x \mid x{=}3\rangle\backslash y \mid y{=}true\rangle \quad\leftrightarrow\quad \langle\langle\rangle \leftarrow x{=}3 \leftarrow y{=}true\rangle$$

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

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, $\langle\!\langle\,\rangle\!\rangle$, 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.

$\langle\!\langle\rangle\!\rangle$	the type of all records. Contains, e.g.: $\langle\rangle$, $\langle x{=}3\rangle$.
$\langle\!\langle\rangle\!\rangle\backslash x$	the type of all records which lack fields of label x. E.g.: $\langle\rangle$, $\langle y{=}true\rangle$, but not $\langle x{=}3\rangle$.
$\langle\!\langle x{:}Int, y{:}Bool\rangle\!\rangle$	the type of all records which have *at least* fields of labels x and y, with values of types Int and $Bool$. E.g.: $\langle x{=}3, y{=}true\rangle$, $\langle x{=}3, y{=}true, z{=}"str"\rangle$, but not $\langle x{=}3, y{=}4\rangle$, $\langle x{=}3\rangle$.
$\langle\!\langle x{:}Int\rangle\!\rangle\backslash y$	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. $\langle x{=}3, z{=}"str"\rangle$, but not $\langle x{=}3, y{=}true\rangle$.

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)[1]. 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.

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

- *Extension* $\langle\!\langle R|x{:}A\rangle\!\rangle$: 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 $\langle r|x{=}a\rangle$ 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 $\langle\!\langle R|x{:}A|y{:}B\rangle\!\rangle$ for $\langle\!\langle\langle\!\langle R|x{:}A\rangle\!\rangle|y{:}B\rangle\!\rangle$.

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

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

$\langle\!\langle\langle\!\langle\rangle\!\rangle\backslash x\|x{:}Int\rangle\!\rangle$	$\leftrightarrow$	$\langle\!\langle x{:}Int\rangle\!\rangle$	extension	
$\langle\!\langle\langle\!\langle x{:}Int\rangle\!\rangle\backslash y\|y{:}Bool\rangle\!\rangle$	$\leftrightarrow$	$\langle\!\langle x{:}Int, y{:}Bool\rangle\!\rangle$		
$\langle\!\langle x{:}Int, y{:}Bool\rangle\!\rangle\backslash y$	$\leftrightarrow$	$\langle\!\langle x{:}Int\rangle\!\rangle\backslash y$	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	

$\langle\!\langle\langle\!\langle\rangle\!\rangle\|x{:}Int\rangle\!\rangle$	invalid extension
$\langle\!\langle\langle\!\langle x{:}Int\rangle\!\rangle\|x{:}Int\rangle\!\rangle$	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.:

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

Then, when considering $\langle\!\langle y{:}Bool\rangle\!\rangle\backslash y$ we actually have the expansion $\langle\!\langle\langle\!\langle\rangle\!\rangle\backslash y|y{:}Bool\rangle\!\rangle\backslash y$. 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.

Remark. It is deceptive to think that every record in $\langle\!\langle R|x{:}A\rangle\!\rangle$ has at least the fields of some record in R (i.e., that $\langle\!\langle R|x{:}A\rangle\!\rangle$ has "more type fields" than R), since $\langle\!\langle R|x{:}A\rangle\!\rangle$ is not necessarily contained in R. For example, if $R{=}\langle\!\langle\rangle\!\rangle\backslash x$ the two collections are incomparable.

Based on this example, one might then think that $\langle\!\langle R\backslash x|x{:}A\rangle\!\rangle$ has more type fields than R, and this is indeed true for $R{=}\langle\!\langle\rangle\!\rangle$. However, in general this fails; for example $R{=}\langle\!\langle\rangle\!\rangle\backslash x$ makes the collections incomparable, and $R{=}\langle\!\langle\langle\!\langle\rangle\!\rangle\backslash x|x{:}A\rangle\!\rangle$ causes the two collections to have the same fields.

It is also deceptive to think that $R\backslash x$ has fewer type fields than R, since R is in general not contained in $R\backslash x$. This containment is true for $R{=}\langle\!\langle\rangle\!\rangle\backslash x$, but false for $R{=}\langle\!\langle\rangle\!\rangle$ where the opposite is true, and $R{=}\langle\!\langle\langle\!\langle\rangle\!\rangle\backslash x|x{:}A\rangle\!\rangle$ makes the two collections incomparable.

These observations might appear to conflict with our previous assertion that positive and negative information always makes things smaller. The assertion is true for normalized record types, but not for arbitrary applications of operators which may later cancel out. We shall study the normalization process in a later section.

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:

$\langle r\|x{=}a\rangle$	where $r{:}\langle\!\langle\rangle\!\rangle\backslash x$
$r\backslash x$	where $r{:}\langle\!\langle\rangle\!\rangle$
$r.x$	where $r{:}\langle\!\langle x{:}A\rangle\!\rangle$ for some A

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:

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

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

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

and can be used as follows:

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

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.

Remark. An alternative syntactic notation, along the lines of [Jategaonkar, Mitchell 1988], could use pattern matching of record parameters:

$$\begin{aligned}&let\ f(\langle rr\backslash y|x=rx\rangle) : \langle\!\langle x{:}Int, y{:}Int\rangle\!\rangle = \\ &\quad \langle rr|x=rx+1|y=0\rangle\end{aligned}$$

Here the actual parameter must match the shape of a record with a field *x* and a collection of remaining components that lack *y*. The variables *rr* and *rx* are bound to the appropriate components and then used in the body of *f*, where *rr* acquires the assumption that it does not contain either *x* or *y* fields. There are some non-trivial details to pattern matching in the presence of subtyping. Since our main objective is to illustrate the fundamental ideas, we choose the simpler syntax.

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:

$\langle\!\langle R\|x{:}A\rangle\!\rangle$	where $R <: \langle\!\langle\rangle\!\rangle\backslash x$
$R\backslash x$	where $R <: \langle\!\langle\rangle\!\rangle$
$R.x$	where $R <: \langle\!\langle x{:}A\rangle\!\rangle$ for some A

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

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

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:

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

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 $\langle\!\langle\langle\!\langle x{:}Int,\ z{:}Bool\rangle\!\rangle\backslash y|y{:}Int\rangle\!\rangle$, 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*:

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

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

Remark. Notice that we have freedom in the typing of the polymorphic function f; for example, we could have chosen the typing:

$$let\ f(R{<:}\langle\!\langle\rangle\!\rangle\backslash x\ y)(r{:}\langle\!\langle R|x{:}Int\rangle\!\rangle) : \langle\!\langle R|x{:}Int|y{:}Int\rangle\!\rangle =$$
$$\langle r \leftarrow x{=}r.x{+}1\,|y{=}0\rangle$$

$$f(\langle\!\langle z{:}Bool\rangle\!\rangle\backslash x\ y)(\langle x{=}3,\ z{=}true\rangle) : \langle\!\langle x{:}Int,\ y{:}Int,\ z{:}Bool\rangle\!\rangle$$

This typing turns out to be incomparable with the previous one; in general we do not seem to have a "best" way of typing an expression. However, we have not studied this aspect of the system carefully.

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:

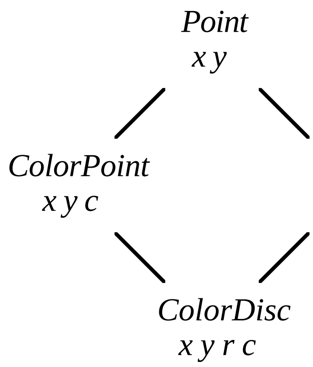
$$let\ Point = \langle\!\langle x{:}Real,\ y{:}Real\rangle\!\rangle$$
$$let\ ColorPoint = \langle\!\langle x{:}Real,\ y{:}Real,\ c{:}Color\rangle\!\rangle$$
$$let\ Disc = \langle\!\langle x{:}Real,\ y{:}Real,\ r{:}Real\rangle\!\rangle$$

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

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:

$$\begin{aligned}
&let\ Point = \langle\!\langle x{:}Real,\ y{:}Real\rangle\!\rangle\\
&let\ ColorPoint = \langle\!\langle Point \leftarrow c{:}Color\rangle\!\rangle\\
&let\ Disc = \langle\!\langle Point \leftarrow r{:}Real\rangle\!\rangle\\
&let\ ColorDisc = \langle\!\langle ColorPoint \leftarrow r{:}Real\rangle\!\rangle
\end{aligned}$$

Note that $\langle\!\langle Point|c{:}Color\rangle\!\rangle$ would not be well-formed here, since members of *Point* may have a *c* label. In section 4.3 we shall examine another way of defining this hierarchy, for example deriving *Point* from *ColorPoint* by “retracting” the *c* field.



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

$$\begin{aligned}
&let\ p{:}Point = \langle x{=}3,\ y{=}4\rangle\\
&let\ cp{:}ColorPoint = \langle p \leftarrow c{=}green\rangle\\
&let\ cd{:}ColorDisc = \langle cp \leftarrow r{=}1\rangle\\
&let\ d{:}Disc = cd\backslash c
\end{aligned}$$

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 $Point\backslash r <: Point$ since *Point* does not contain *r*, but $Point\backslash y$ is incomparable with *Point* since *Point* requires $y{:}Int$ while $Point\backslash y$ 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:

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

The overriding operator here preserves the additional fields of r.

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

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

In words, we expect *update* to be a function from R to R, for any subtype R of $\langle\!\langle b{:}Bool\rangle\!\rangle$. 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[2]:

$$\begin{aligned}&true : True <: Bool\\ &not : Bool \rightarrow Bool \qquad \text{(alternatively, } not : \forall(A{<:}Bool)A{\rightarrow}Bool)\end{aligned}$$

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

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:

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

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

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 $\forall(B{<:}A)\ B \rightarrow B$ turns out to contain only the identity function on A in many natural semantic models, such as [Bruce, Longo 1990]. For example take $A{=}Int$ and let the subranges $[n..m]$ be subtypes of *Int*. Then any function of type $\forall(B{<:}Int)\ B$

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

$\rightarrow B$ can be instantiated to $[n..n] \rightarrow [n..n]$, 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:

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

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 1985], 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*:

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

Soundness is then recovered by requiring that assignable fields be both covariant and contravariant. Hence, *True* <: *Bool* does not imply ⟪*var b*:*True*⟫ <: ⟪*var b*:*Bool*⟫, 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:

$\vdash E\ env$	E is an environment
$E \vdash A\ type$	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.

(ENV1) $$\frac{}{\vdash \emptyset\ env}$$

(ENV2) $$\frac{X \notin dom(E)}{\vdash E, X\ env}$$

(ENV3) $$\frac{E \vdash A\ type \quad X \notin dom(E)}{\vdash E, X{<:}A\ env}$$

(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 ϕ 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., ϕ, $X{<:}A$, $X{<:}B$) but this would push us into the more general discipline of *conjunctive types*.

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

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

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 assumptions 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.

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

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 **(VAR2)** in the previous section); an extension $\langle\!\langle R|x{:}A\rangle\!\rangle$ 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 $R{<:}\langle\!\langle S|x{:}A\rangle\!\rangle$ 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 (these are listed in section 3.7). 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.

(TE1)

$$\frac{E \vdash R <: \langle\!\langle\rangle\!\rangle \backslash xy \quad E \vdash A,B\ type \quad x \neq y}{E \vdash \langle\!\langle \langle\!\langle R|x{:}A \rangle\!\rangle | y{:}B \rangle\!\rangle \leftrightarrow \langle\!\langle \langle\!\langle R|y{:}B \rangle\!\rangle | x{:}A \rangle\!\rangle}$$

Similarly, we can reorder restrictions. Moreover, a double restriction $R\backslash xx$ 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)

$$\frac{E \vdash R <: \langle\!\langle\rangle\!\rangle \backslash x \quad E \vdash A\ type}{E \vdash \langle\!\langle R|x{:}A \rangle\!\rangle \backslash x \leftrightarrow R}$$

(TE6)

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

Note however that in a situation like $\langle\!\langle R\backslash x|x{:}A \rangle\!\rangle$ 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)

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

(TE8)

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

(TE4)

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

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, *et al.* 1989] [Curien, Ghelli 1992], 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 $\langle\!\langle R|x{:}A\rangle\!\rangle$, (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)
$$\frac{E \vdash R <: \langle\!\langle\rangle\!\rangle\backslash x \quad E \vdash A\ type}{E \vdash \langle\!\langle R|x{:}A\rangle\!\rangle <: \langle\!\langle\rangle\!\rangle}$$

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

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

Remark. Although this rule looks innocent, it hides some interesting subtlety in its assumption. Let us analyze $R<:S$ by cases.

The cases when R and S are themselves restrictions (either of x or of some other variable) are straightforward. Similarly simple are the cases when R and S are matching extensions, both of them either containing or not containing an x field.

Suppose however that R has a positive x field and S does not, for example $R=\langle\!\langle T|x{:}A\rangle\!\rangle$ and $S=T$. In that case, if we had $R<:S$ we would erroneously conclude that $R\backslash x = \langle\!\langle T|x{:}A\rangle\!\rangle\backslash x \leftrightarrow T <: T\backslash x = S\backslash x$ (which is false for $T=\langle\!\langle\rangle\!\rangle$).

Fortunately there was a flaw in this argument; the assumption for **(S4)** requires $R = \langle\!\langle T|x{:}A\rangle\!\rangle <: T = S$, but this is false (for $T=\langle\!\langle\rangle\!\rangle\backslash x$). Note also that taking $R=\langle\!\langle T\backslash x|x{:}A\rangle\!\rangle$ and $S=T$ leads to a similar contradiction for $T=\langle\!\langle\rangle\!\rangle\backslash x$.

A legal instance of the assumption is $R = \langle\!\langle \langle\!\langle\rangle\!\rangle \backslash x | x{:}A \rangle\!\rangle <: \langle\!\langle\rangle\!\rangle = S$, from which we conclude that $R\backslash x = \langle\!\langle \langle\!\langle\rangle\!\rangle \backslash x | x{:}A \rangle\!\rangle \backslash x \leftrightarrow \langle\!\langle\rangle\!\rangle \backslash x <: \langle\!\langle\rangle\!\rangle \backslash x = S\backslash x$, which is correct.

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 $R <: \langle\!\langle S|x{:}B \rangle\!\rangle$. If R can be converted to the form $R=\langle\!\langle R'|x{:}A \rangle\!\rangle$, 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)
$$\frac{E \vdash R <: \langle\!\langle S|x{:}A \rangle\!\rangle <: \langle\!\langle\rangle\!\rangle}{E \vdash R.x <: A}$$

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 $\langle\!\langle R\backslash x|x{:}R.x \rangle\!\rangle$, 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:

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

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:

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

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:

(I1)
$$\frac{\vdash E\ env}{E \vdash \langle\rangle\backslash x_1..x_n : \langle\!\langle\rangle\!\rangle}$$

(I2)
$$\frac{E \vdash \langle\rangle\backslash x_1..x_n : R <: \langle\!\langle\rangle\!\rangle}{E \vdash \langle\rangle\backslash x_1..x_n : R\backslash y}$$

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

(I3)
$$\frac{E \vdash r{:}R{<:}\langle\!\langle\rangle\!\rangle\backslash x \quad E \vdash a{:}A}{E \vdash \langle r|x{=}a\rangle : \langle\!\langle R|x{:}A\rangle\!\rangle}$$

(E1)
$$\frac{E \vdash r{:}R{<:}\langle\!\langle\rangle\!\rangle}{E \vdash r\backslash x : R\backslash x}$$

(E2)
$$\frac{E \vdash r{:}\langle\!\langle R|x{:}A\rangle\!\rangle{<:}\langle\!\langle\rangle\!\rangle}{E \vdash r.x : A}$$

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

$$\begin{array}{ll}
 & \underline{E \vdash r{:}R{<:}\langle\!\langle S|x{:}A\rangle\!\rangle{<:}\langle\!\langle\rangle\!\rangle} \\
\textbf{(S6)} & \underline{E \vdash R{<:}\langle\!\langle R\backslash x|x{:}R.x\rangle\!\rangle} \\
\textbf{(G3)} & \underline{E \vdash r{:}\langle\!\langle R\backslash x|x{:}R.x\rangle\!\rangle} \\
\textbf{(E2)} & E \vdash r.x : R.x
\end{array}$$

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 $\langle\rangle\backslash x \leftrightarrow \langle\rangle$. 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 1985]), 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)
$$\frac{E \vdash A \leftrightarrow B}{E \vdash B \leftrightarrow A}$$

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

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

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

$E \vdash b \leftrightarrow a : A \qquad\qquad E \vdash a \leftrightarrow c : A$

Formation

(F1)...(F4)

Subtyping

(S1)...(S6)

Introduction/Elimination

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

Type Congruence

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

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

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

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

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

Type Equivalence

(TE1)...(TE8)

Value Congruence

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

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

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

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

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

Value Equivalence

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

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

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

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

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

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

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

(VE8)

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

(VE9)

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

(VE10)

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

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)

$$\frac{E \vdash R{<:}\langle\!\langle S|x{:}A\rangle\!\rangle{<:}\langle\!\langle\rangle\!\rangle}{E \vdash R \leftrightarrow \langle\!\langle R\backslash x|x{:}R.x\rangle\!\rangle}$$

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)

$$\frac{E \vdash R \leftrightarrow \langle\!\langle S|x{:}A\rangle\!\rangle{<:}\langle\!\langle\rangle\!\rangle}{E \vdash R.x \leftrightarrow A}$$

(VC5b)

$$\frac{E \vdash r \leftrightarrow s : \langle\!\langle R|x{:}A\rangle\!\rangle{<:}\langle\!\langle\rangle\!\rangle}{E \vdash r.x \leftrightarrow s.x : A}$$

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*.

Proof

Simple simultaneous induction on derivations, with **(F1)** as the base case.

□

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.*

Proof

Show (1) and (2) simultaneously by induction on derivations. The hardest case is **(TE1)**. The next hardest is **(TE8)**. All the others are substantially simpler. We prove **(TE1)** below and leave the remaining cases to the reader.

To prove (1) for **(TE1)**, we assume $E \vdash R{<:}\langle\!\langle\rangle\!\rangle\backslash xy$ and $E \vdash A,B$ type. Using **(S2)** and **(S4)** we may derive $E \vdash \langle\!\langle\rangle\!\rangle\backslash xy{<:}\langle\!\langle\rangle\!\rangle\backslash x$ and so by transitivity and **(F2)** we have $E \vdash \langle\!\langle R|x{:}A\rangle\!\rangle$ *type*. The next goal is to show that $\langle\!\langle R|x{:}A\rangle\!\rangle$ is a subtype of $\langle\!\langle\rangle\!\rangle\backslash y$. Using **(S2)** and **(S4)** we have $E \vdash R{<:}\langle\!\langle\rangle\!\rangle\backslash y$ by transitivity, and so by **(TE2)**, $E \vdash R\backslash y \leftrightarrow R$. The type congruence rules give $E \vdash \langle\!\langle R|x{:}A\rangle\!\rangle \leftrightarrow \langle\!\langle R\backslash y|x{:}A\rangle\!\rangle$. By **(TE6)** and transitivity we now have $E \vdash \langle\!\langle R|x{:}A\rangle\!\rangle \leftrightarrow \langle\!\langle R|x{:}A\rangle\!\rangle\backslash y$. From **(S1)** and the original hypotheses, it is easy to show $E \vdash \langle\!\langle R|x{:}A\rangle\!\rangle <: \langle\!\langle\rangle\!\rangle$ and so by **(S4)**, $E \vdash \langle\!\langle R|x{:}A\rangle\!\rangle\backslash y <: \langle\!\langle\rangle\!\rangle\backslash y$. This allows us to derive $E \vdash \langle\!\langle R|x{:}A\rangle\!\rangle <: \langle\!\langle\rangle\!\rangle\backslash y$, from which we may finally obtain $E \vdash \langle\!\langle\langle\!\langle R|x{:}A\rangle\!\rangle|y{:}B\rangle\!\rangle$ *type.*

The proof of *E* ⊢ ⟪⟪*R*|*y*:*B*⟫|*x*:*A*⟫ *type* is similar.

□

Sample derivations

We show the main steps of some derivations that can be carried out in this system, assuming rules for typing basic constants.

The first example simply builds a record of two fields, with its natural type.

```
(I1)            ⟨⟩ : ⟪⟫
(E1)            ⟨⟩\x : ⟪⟫\x          (const)   3 : Int
(I3)            ⟨⟨⟩\x|x=3⟩ : ⟪⟪⟫\x|x:Int⟫
(E1)            ⟨⟨⟩\x|x=3⟩\y : ⟪⟪⟫\x|x:Int⟫\y          (const)      true : Bool
(I3)            ⟨⟨⟨⟩\x|x=3⟩\y|y=true⟩ : ⟪⟪⟪⟫\x|x:Int⟫\y|y:Bool⟫
(def)           ⟨x=3, y=true⟩ : ⟪x:Int, y:Bool⟫
```

Next, we derive a non-trivial type inclusion. To construct record types of different lengths on the two sides of <:, we start with the basic asymmetry of **(S1)** and we build up symmetrically from there (there is no more direct way).

```
(G1)            ⟪⟫ <: ⟪⟫
(S4)            ⟪⟫\x <: ⟪⟫\x
(S1)            ⟪⟪⟫\x|x:Int⟫ <: ⟪⟫
(S4)            ⟪⟪⟫\x|x:Int⟫\y <: ⟪⟫\y          (G1)     Bool <: Bool
(S3)            ⟪⟪⟪⟫\x|x:Int⟫\y|y:Bool⟫ <: ⟪⟪⟫\y|y:Bool⟫
(def)           ⟪x:Int, y:Bool⟫ <: ⟪y:Bool⟫
```

Now we show that a given record lacks a given label. This time the key rule is **(I2)**. Some type equivalence rules are used to rearrange the type into a standard form.

```
(I1)            ⟨⟩ : ⟪⟫
(I2)            ⟨⟩ : ⟪⟫\y            (S2)     ⟪⟫\y <: ⟪⟫
(E1)            ⟨⟩\x : ⟪⟫\y\x        (S4)     ⟪⟫\y\x <: ⟪⟫\x        (const)   3 : Int
(I3)            ⟨⟨⟩\x|x=3⟩ : ⟪⟪⟫\y\x|x:Int⟫
(TE3,TC3,G1,G3) ⟨⟨⟩\x|x=3⟩ : ⟪⟪⟫\x\y|x:Int⟫
(TE6,G1,G3)     ⟨⟨⟩\x|x=3⟩ : ⟪⟪⟫\x|x:Int⟫\y
(def)           ⟨x=3⟩ : ⟪x:Int⟫\y
```

Finally, we show that by removing a label we obtain a subtype. The basic asymmetry here is provided by **(S2)**.

```
(G1)            ⟪⟫ <: ⟪⟫
(S2)            ⟪⟫\y <: ⟪⟫
```

$$\begin{array}{ll}
\textbf{(S4)} & \langle\!\langle\rangle\!\rangle\backslash y\backslash x <: \langle\!\langle\rangle\!\rangle\backslash x \qquad \textbf{(G1)} \quad Int <: Int \\ \hline
\textbf{(S3)} & \langle\!\langle\langle\!\langle\rangle\!\rangle\backslash y\backslash x|x{:}Int\rangle\!\rangle <: \langle\!\langle\langle\!\langle\rangle\!\rangle\backslash x|x{:}Int\rangle\!\rangle \\ \hline
\textbf{(TE3,TC2,G1,G2)} & \langle\!\langle\langle\!\langle\rangle\!\rangle\backslash x\backslash y|x{:}Int\rangle\!\rangle <: \langle\!\langle\langle\!\langle\rangle\!\rangle\backslash x|x{:}Int\rangle\!\rangle \\ \hline
\textbf{(TE6,G1,G2)} & \langle\!\langle\langle\!\langle\rangle\!\rangle\backslash x|x{:}Int\rangle\!\rangle\backslash y <: \langle\!\langle\langle\!\langle\rangle\!\rangle\backslash x|x{:}Int\rangle\!\rangle \\ \hline
\textbf{(def)} & \langle\!\langle x{:}Int\rangle\!\rangle\backslash y <: \langle\!\langle x{:}Int\rangle\!\rangle
\end{array}$$

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):

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

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 G_1 , G_2 , ... corresponding to the type symbols G_1 , G_2 , ... such that $G_i \subseteq G_j$ if $G_i <: G_j$ is a subtyping axiom.

For simplicity, we assume that no element of any 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 sections 3.9 and 3.10.

Since ⟪⟫ 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 < ω. 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 R_i of records of rank i, and set V_i of values of rank i, as follows:

$$\begin{array}{lcllcl} V_0 & = & \bigcup_j G_j & V_{i+1} & = & R_i \cup V_i \\ R_0 & = & L \rightharpoonup_{\text{fin}} V_0 & R_{i+1} & = & L \rightharpoonup_{\text{fin}} V_{i+1} \end{array}$$

$$\begin{array}{lcll} R & = & \bigcup_{i<\omega} R_i & \text{the set of } \textit{records} \\ V & = & \bigcup_{i<\omega} V_i & \text{the set of } \textit{values} \end{array}$$

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

$$R \;=\; (L \rightharpoonup_{\text{fin}} V) \;\subseteq\; V$$

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

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

$$\emptyset \quad = \quad \lambda y{\in}L.\ \uparrow$$

$r\text{-}x \;=_{\text{def}}\; \lambda y{\in}L.\ \textit{if } y{=}x \textit{ then } \uparrow \textit{ else } r(y)$
provided $r{\in}R$ and $x{\in}L$

$r[x{=}a] \;=_{\text{def}}\; \lambda y{\in}L.\ \textit{if } y{=}x \textit{ then } a \textit{ else } r(y)$
provided $r{\in}R$, $x{\in}L$, $a{\in}V$, and $x{\notin}dom(r)$.

$r(x)$ is well-defined,
provided $r{\in}R$, $x{\in}L$, and $x{\in}dom(r)$.

Lemma 3.8.1:

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

Proof

(1) The empty function is a finite function.
(2) If $r{\in}R$ then $r\text{-}x$ remains a finite partial function in R.
(3) Suppose $r{\in}R$ with $x \notin dom(r)$, and $a{\in}V$.
Then $r[x{=}a]$ is well-defined (is a function) and belongs to R.

(4) If $r \in R = L \rightharpoonup_{\text{fin}} V$ and $r(x)$ is defined then $r(x) \in V$.

□

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:

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

Lemma 3.8.2:

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

Proof

(1) If $R \subseteq R$, then $R\text{-}x = \{r\text{-}x \mid r \in R\} \subseteq R$, by 3.8.1.

(2) If $R \subseteq R\text{-}x$, then R is a set of functions $r \in L \rightharpoonup_{\text{fin}} V$ with $x \notin dom(r)$.
Hence for any $A \subseteq V$, $R[x{:}A] = \{r[x{=}a] \mid r \in R, a \in A\} \subseteq R$, by 3.8.1.

(3) If $R \subseteq S[x{:}A]$, then for any $r \in R$, $r \in S[x{:}A] = \{s[x{=}a] \mid s \in S, a \in A\}$;
so that $r(x) \in A$. Hence $R(x) = \{r(x) \mid r \in R\} \subseteq A \subseteq V$.

□

Interpretation of judgments

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

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

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.

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

$E \vdash a \leftrightarrow b : A \qquad \approx \qquad a_\rho = b_\rho \in A_\rho \subseteq V$, for every ρ satisfying E.

Type and value expressions are interpreted using:

$$
\begin{array}{lll}
\langle\!\langle\rangle\!\rangle & \approx & R \\
R\backslash x & \approx & R\text{-}x \\
\langle\!\langle R|x{:}A\rangle\!\rangle & \approx & R[x{:}A] \\
R.x & \approx & R(x) \\
\\
\langle\rangle & \approx & \emptyset \\
r\backslash x & \approx & r\text{-}x \\
\langle r|x{=}a\rangle & \approx & r[x{=}a] \\
r.x & \approx & r(x)
\end{array}
$$

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.

Proof

See appendix.

□

3.9 A construction giving $R = \langle\!\langle R\backslash x|x{:}R.x\rangle\!\rangle$

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)

$$\frac{E \vdash R <: \langle\!\langle S|x{:}A\rangle\!\rangle <: \langle\!\langle\rangle\!\rangle}{E \vdash R \leftrightarrow \langle\!\langle R\backslash x|x{:}R.x\rangle\!\rangle}$$

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)**:

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

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-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, $R \neq \langle\!\langle R\backslash x | x{:}R.x \rangle\!\rangle$ 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 $R = \langle\!\langle R\backslash x | x{:}R.x \rangle\!\rangle$ says that R is the product of $R\backslash x$ and $R.x$.

In this section 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 $R = \langle\!\langle R\backslash x | x{:}R.x \rangle\!\rangle$ 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 V as types. In this way we eliminate sets of records which violate **(TE9)**.

We use a value space satisfying:

$$R \;=\; (L \rightharpoonup_{\mathrm{fin}} V) \;\subseteq\; V$$

constructed as in section 3.8. Then for each natural number i, we define the collection T_i of subsets of V which we wish to consider types of stage i. At the first stage, we may select any subsets of V, provided we include the given ground types G_j . For definiteness, let us take:

$$T_0 \;=\; \{G_1 , G_2 , \dots \}$$

We now define record types over preceding types. At each stage we take all record types defined by a finite set of labeled component types, and a finite set of absent labels. Each component type must belong to the preceding stage.

This construction may be clarified using an auxiliary definition. Suppose $P{:}\, L \rightharpoonup_{\mathrm{fin}} T_i$ is a finite partial function from labels to types at stage i, and $N \subseteq_{\mathrm{fin}} L$ is a finite set of labels disjoint from the domain of P. Then the set $R^{P,N}$ of records with components present according to P and components absent according to N is defined by:

$$R^{P,N} \;=\; \{r{\in}R \mid \forall x{\in}L.\, (P(x){\downarrow} \;\supset\; r(x){\in}P(x)) \;\wedge\; (x{\in}N \;\supset\; r(x){\uparrow})\}$$

We define the set of record types at stage $i+1$ to be the set of all $R^{P,N}$ for suitable "present" function P and "absent" set N:

$$T_{i+1} \;=\; \{R^{P,N} \mid P{:}\, L \rightharpoonup_{\mathrm{fin}} T_i \;\wedge\; N \subseteq_{\mathrm{fin}} L \;\wedge\; dom(P) \cap N = \emptyset\} \;\cup\; T_i$$

Note that $R = R^{\emptyset,\emptyset}$ belongs to every T_{i+1}.

The collection T of all types is defined by:

$$T \;=\; \textstyle\bigcup_{i < \omega} T_i$$

As we have defined T, the set V of all values is not a type. However, it is possible to include V in T_0 if desired.

It is natural to consider any set of records $R^{P,N}$ with $P{:}\, L \rightharpoonup_{\mathrm{fin}} T$ and $N \subseteq_{\mathrm{fin}} L$, as a "record type" over V. Define RT to be the collection of all record types:

$$RT \;=_{\mathrm{def}}\; \{R^{P,N} \mid P{:}\, L \rightharpoonup_{\mathrm{fin}} T ,\, N \subseteq_{\mathrm{fin}} L, \text{ and } dom(P) \cap N = \emptyset\}$$

Note that $R^{\emptyset,\emptyset} = \bigcup RT$, so RT has a maximal element. We may show that T is precisely the union of T_0 and the record types over V; that is $T = T_0 \cup RT$.

Lemma 3.9.1:

If P: $L \rightharpoonup_{\text{fin}} T$ and $N \subseteq_{\text{fin}} L$ with $dom(P) \cap N = \emptyset$, then $R^{P,N} \in T$.

That is, $RT \subseteq T$.

Proof

Suppose P: $L \rightharpoonup_{\text{fin}} T$ and $N \subseteq_{\text{fin}} L$. Since the domain of P is finite, there is some i with P: $L \rightharpoonup_{\text{fin}} T_i$. Hence, $R^{P,N} \in T_{i+1} \subseteq T$.

□

In this model we will interpret all judgments as before, except that type variables and type expressions must denote elements of T. Since we consider only elements of T as types, we define the relation $A \subseteq: B$ (*A semantic subtype* of B) as:

$A \subseteq: B$ iff $A \subseteq B$ and $A,B \in T$

By the simplifying assumption in section 3.9 that no ground type contains records, we know that every subtype of R will be an element of RT. If we had not made this assumption, then we might have some subtype of R which "accidentally" could cause **(TE9)** to fail.

We may show that for any non-empty $R \in RT$, a function P and set N with $R = R^{P,N}$ are determined uniquely.

Lemma 3.9.2:

Let $R \in RT$ be non-empty. Then $R = R^{P,N}$ where:

$dom(P) = \{x \in L \mid \forall r \in R.\ r(x)\downarrow\}$,

$N = \{x \in L \mid \forall r \in R.\ r(x)\uparrow\}$, and

$P(x) = R(x)$ for all $x \in dom(P)$

Proof

Suppose $R \in RT$ is non-empty and let $r_0 \in R$.

We know that $R = R^{P,N}$ for some P,N.

(1) By construction of $R^{P,N}$ we have $\forall r \in R.\ dom(P) \subseteq dom(r)$.

Moreover, if $\forall r \in R.\ r(x)\downarrow$, then $x \in dom(P)$, since $x \notin dom(P)$ implies $r_0\text{-}x \in R$ and $(r_0\text{-}x)(x)\uparrow$. Consider the function f defined by:

$f(x) = r_0(x)$ if $\forall r \in R.\ r(x)\downarrow$, and $\uparrow$ otherwise

This function belongs to R, and $dom(f) = \{x \in L \mid \forall r \in R.\ r(x)\downarrow\} \subseteq dom(P)$.

Hence $dom(P) = dom(f) = \{x \in L \mid \forall r \in R.\ r(x)\downarrow\}$.

(2) By construction of $R^{P,N}$ we have $\forall r \in R.\ N \subseteq \uparrow(r) =_{\text{def}} \{x \in L \mid r(x)\uparrow\}$.

Moreover, if $\forall r \in R.\ r(x)\uparrow$, then $x \in N$ (since $x \notin N$ implies either $r_0(x)\downarrow$ or $(r_0[x=a])(x)\downarrow$ for an appropriately chosen $r_0[x=a] \in R$).

Choose r_x from $R_x =_{\text{def}} \{r \in R \mid r(x)\downarrow\}$ whenever $R_x \neq \emptyset$, and define:

$g(x) = \uparrow$ if $\forall r \in R.\ r(x)\uparrow$, and $r_x(x)$ otherwise

This function belongs to R and $\uparrow(g) = \{x \in L \mid \forall r \in R.\ r(x)\uparrow\} \subseteq N$.

Hence, $N = \uparrow(g) = \{x \in L \mid \forall r \in R.\ r(x)\uparrow\}$.

(3) Assume $x \in dom(P)$.

$$R(x) = R^{P,N}(x) = \{r(x) \mid r \in R\ ,\ \forall y \in L.\ r(y) \in P(y)\} \quad (\text{since } x \notin N)$$
$$= \{r(x) \mid r \in R\ ,\ r(x) \in P(x)\} = \{a \in V \mid a \in P(x)\} = P(x)$$

□

This allows us to write each non-empty record type $R \in RT$ as $R^{P,N}$ without ambiguity. The lemma also demonstrates that whenever $R(x)$ is defined, $R(x) = R^{P,N}(x) = P(x) \in T$ is a type.

It is now straightforward to show that the record types are closed under restriction (R-x) and extension ($R[x{:}B]$):

Lemma 3.9.3:

If $R = R^{P,N}$ is any record type, then $R\text{-}x = R^{P',N'}$, where

$P' = P - \{<x \mapsto P(x)>\}$ if $P(x)\downarrow$, and P otherwise.

$N' = N \cup \{x\}$

Proof

Straightforward.

□

Lemma 3.9.4:

If $R = R^{P,N}$ with $x \in N$, and $B \in T$, then $R[x{:}B] = R^{P',N'}$, with:

$P' = P \cup \{<x \mapsto B>\}$

$N' = N\text{-}\{x\}$

Proof

By definition, $R[x{:}B] = \{r[x{=}b] \mid r \in R, b \in B\}$. It is easy to check that every $r[x{=}b]$ belongs to $R^{P',N'}$ and conversely.

□

The semantic subtyping relation on record types $R \subseteq: R'$ is now determined by the present and absent information.

Lemma 3.9.5:

$R^{P,N} \subseteq: R^{P',N'}$ iff

$\forall x \in dom(P').\ P(x)\downarrow \wedge P(x) \subseteq: P'(x)$

$N' \subseteq N$

Proof

Assume $R^{P,N} \subseteq: R^{P',N'}$.

It is easy to check that $N' \subseteq N$ by the definition of $R^{P,N}$.

Similarly, if $P'(x)\downarrow$ then we must have $P(x)\downarrow \wedge P(x) \subseteq P'(x)$.

By definition $P(x)$ and $P'(x)$ are types.

The converse is straightforward.

□

Since the point of this model construction is to give $R = (R\text{-}x)[x{:}R(x)]$ for every record type R with $R(x)\downarrow$, we must also prove this equation. Given the preceding lemmas, the proof is almost immediate.

Lemma 3.9.6:

Let $R \in RT$ be a record type with $r(x)\downarrow$ for all $r \in R$.

Then $R = (R\text{-}x)[x{:}R(x)]$.

Proof

We know $R = R^{P,N}$ for some finite function P and finite set N.

By preceding lemmas, we also have:

$R\text{-}x \quad = \quad R^{P',N'}$

$(R\text{-}x)[x{:}R(x)] \quad = \quad R^{P'',N''}$

with $P' = P - \{<x \mapsto R(x)>\}$, $N' = N \cup \{x\}$

and $P'' = P' \cup \{<x \mapsto R(x)>\}$, $N'' = N' - \{x\}$.

Since $P'' = P$ and $N'' = N$, it follows that $R = (R\text{-}x)[x{:}R(x)]$.

□

Finally, we have the soundness theorem. System *S2* is system *S1* of Theorem 3.8.3 plus the rule **(TE9)**.

Theorem 3.9.7 (soundness):

The inference rules of system *S2* are sound with respect to the interpretation of judgments given above.

Proof

See appendix.

□

3.10 An extensional model construction

The following inference rule 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 reason for adopting 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 $r,s : \langle\!\langle x_1{:}A_1 , ... , x_k{:}A_k\rangle\!\rangle$, we have $r \leftrightarrow s : \langle\!\langle x_1{:}A_1 , ... , x_k{:}A_k\rangle\!\rangle$ iff $r.x_i \leftrightarrow s.x_i : A_i$ for $i = 1...k$. The straightforward proof of this uses $r\backslash x_1...x_k \leftrightarrow s\backslash x_1...x_k : \langle\!\langle\rangle\!\rangle$ and the value congruence rules.

Recall that in the previous models a record type was simply a set of records, and equality of records was independent of the type. Therefore, any two distinct records would be unequal elements of $\langle\!\langle\rangle\!\rangle$, causing **(VC1b)** to fail.

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

We begin with a value space V satisfying:

$$R \;=\; (L \rightharpoonup_{\text{fin}} V) \;\subseteq\; V$$

constructed as in section 3.8, and define types as certain *partial equivalence relations* (abbreviated *PER*'s) over V (see [Longo, Moggi 1991]). A PER is a binary relation which is symmetric and transitive, but not necessarily reflexive. An element of a type is defined as an equivalence class of values in the PER.

Subtyping is based on set containment of partial equivalence relations, as in [Bruce, Longo 1990], except that we consider only certain PER's as types.

The type of all records ⟪⟫ is interpreted by the PER $R \times R$. This type has only one element since there is a single equivalence class in $R \times R$: while ⟪⟫ contains all records, all records are equivalent in ⟪⟫ (hence **(VC1b)** holds).

The three operations on record types are defined as follows:

- If R is a PER on R with $r(x)\uparrow$ for every record rRr, and A is a PER on V, then $R[x{:}A]$ is the relation on R given by:

  $$r\ R[x{:}A]\ s \quad \text{iff} \quad r\text{-}x\ R\ s\text{-}x \ \text{ and } \ r(x)\ A\ s(x)$$

  In writing $r(x)\ A\ s(x)$ we imply that $r(x)\downarrow$ and $s(x)\downarrow$.

- If R is a PER on R , we define the relation R-x by:

  $$R\text{-}x \;=_{\text{def}}\; \{<r\text{-}x,\, s\text{-}x> \mid rRs\}$$

- If R is a PER on R , with $r(x)\downarrow$ whenever rRr, we define the relation $R(x)$ by:

  $$R(x) \;=_{\text{def}}\; \{<r(x),\, s(x)> \mid rRs\}$$

It is easy to show that under the hypotheses above, $R[x{:}A]$ is a partial equivalence relation on R . However, R-x and $R(x)$ are not necessarily transitive. This will not cause any problems, as it turns out, since by restricting the class of record types to some collection satisfying **(TE9)**, R-x and $R(x)$ are guaranteed to be types (and hence PER's).

The types over V will be defined in stages, as before. We begin with some collection:

$$T_0 \;=\; \{E_1 , E_2 , \ldots \}$$

of partial equivalence relations over V that do not relate any records to themselves. A typical choice would be to begin with the identity relations on the ground types G_1 , G_2 , ...

Given any finite partial map P from L to PER's over V and a set $N \subseteq_{\text{fin}} L$ disjoint from the the domain of P, we define the PER $R^{P,N}$ over R by:

$$r\ R^{P,N}\ s \quad \text{iff} \quad \forall x{\in}L.\ (P(x){\downarrow}\ \supset\ r(x)\ P(x)\ s(x))\ \wedge\ (x{\in}N\ \supset\ r(x){\uparrow} \wedge s(x){\uparrow})$$

Note the similarity to $R^{P,N}$ for subsets of V; if we represent a subset $S{\subseteq}V$ by the PER $(S{\times}S) \subseteq (V \times V)$, the two definitions coincide. It is easy to see that if each $P(x)$ is a PER, then so is $R^{P,N}$.

Following the earlier definition of record types in stages, we define:

$$T_{i+1} \ = \ \{R^{P,N} \mid P{:}L \rightharpoonup_{\mathrm{fin}} T_i \ \wedge \ N{\subseteq_{\mathrm{fin}}}L \ \wedge \ dom(P){\cap}N = \emptyset\} \ \cup \ T_i$$

and let:

$$T \ = \ \bigcup_{i < \omega} T_i$$

This construction has much the same character as the previous non-extensional one, although we have the added complication of establishing that R-x and $R(x)$ (when defined) are PER's whenever $R{\in}T$. Since every $R{\in}T$ is easily seen to be a PER, we will do this by showing R-$x{\in}T$ and $R(x){\in}T$.

It is easy to prove Lemma 3.9.1 for this model, showing that we need not consider stages of the construction in later arguments.

Lemma 3.10.1:
If $P{:}\ L \rightharpoonup_{\mathrm{fin}} T$ and $N \subseteq_{\mathrm{fin}} L$ with $dom(P){\cap}N = \emptyset$, then $R^{P,N} \in T$.

Define the collection of all record types by $RT = \{R^{P,N}\}$.
Subtyping is interpreted as before, with:

$$A \subseteq{:}\ B \quad \text{iff} \quad A \subseteq B \text{ and } A,B \in T$$

We now use present functions and absent sets to show that for every $R \in RT$, we have R-$x{\in}T$ and $R(x){\in}T$ if $r(x){\downarrow}$ for every rRr.

Lemma 3.10.2:
If $R \in RT$, then R-$x{\in}T$.
If $R \in RT$ with $r(x){\downarrow}$ whenever rRr, then $R(x){\in}T$.
Proof
The lemma is trivial if $R = \emptyset$, hence we assume $R \neq \emptyset$.
(1) Let $R{=}R^{P,N}$. Then R-$x = R^{P',N'}$ with $P' = P - \{<x{\mapsto}P(x)>\}$ and $N' = N \cup \{x\}$. To see this, suppose $r\ R$-$x\ s$. Then there must be records $r',s'{\in}R$ with $r'Rs'$ and $r{=}r'$-x, $s{=}s'$-x.
Since $P'(y){\downarrow}\ \supset\ r(y)\ P(y)\ s(y)$ and $y{\in}N'\ \supset\ r(y){\uparrow} \wedge s(y){\uparrow}$, it follows that $r\ R^{P',N'}\ s$.
To show the converse, we assume $r\ R^{P',N'}\ s$ and note that since $R \neq \emptyset$, there must be some $b{\in}V$ with $b\ P(x)\ b$. It is easy to see that $r[x{=}b]\ R\ s[x{=}b]$, and so $r\ R$-$x\ s$.
(2) We now assume $r(x){\downarrow}$ whenever rRr. Since $R{=}R^{P,N}$, we have $P(x){\in}T$. It remains to show that $R(x){=}P(x)$. If $a\ R(x)\ b$, then there exist r and s with rRs and $a{=}r(x)$, $b{=}s(x)$.
By definition of $R^{P,N}$ it follows that $a\ P(x)\ b$.

For the converse, we assume $a\ P(x)\ b$; since $R\neq\emptyset$, there exist r and s with $r\ R^{P,N}\ s$ and $r(x)=a$, $s(x)=b$. Hence $a\ R(x)\ b$.

□

Lemma 3.10.3:

If $R \in RT$ with $r(x)\uparrow$ whenever rRr, and $B\in T$, then $R[x{:}B]\in T$.

Proof

The lemma is trivial if $R=\emptyset$. Otherwise, we let $R=R^{P,N}$ and show that $R[x{:}B]=R^{P',N'}$ with $P' = P \cup \{<x\mapsto B>\}$ and $N' = N\text{-}\{x\}$.

This is straightforward.

□

It is now an easy matter to show analogs of Lemma 3.9.2 and Lemma 3.9.6. These conclude the basic properties of the construction. System *S3* is system *S1* of Theorem 3.8.3 plus the rules **(TE9)** and **(VC1b)**.

Theorem 3.10.4 (soundness):

The inference rules of system *S3* are sound for the PER model construction.

Proof

See appendix.

□

3.11 The update operator

Extensional models are useful to characterize a natural form of record update, here denoted by $r.x :\text{-}\ a$ for functional update. The discussion is also relevant to the typing of imperative update, $r.x := a$, although our models do not directly capture side-effects.

The functional update operator cannot be introduced by a simple definition. We want:

$$r.x :\text{-}\ a \qquad =_{\text{def}} \qquad \langle r\backslash x|x{=}a\rangle$$

but only provided that $r.x$ exists, and that $r.x :\text{-}\ a$ does not modify the type of the x field. Sufficient assumptions are that $r{:}R{<:}\langle\!\langle\rangle\!\rangle$ and $a{:}R.x$; then we can derive the following typing:

$$\begin{array}{ll}
 & \underline{E\vdash r{:}R{<:}\langle\!\langle\rangle\!\rangle} \\
\textbf{(E1)} & \underline{E\vdash\ r\backslash x : R\backslash x \qquad E\vdash a{:}R.x} \\
\textbf{(I1)} & \underline{E\vdash\ \langle r\backslash x|x{=}a\rangle : \langle\!\langle R\backslash x|x{:}R.x\rangle\!\rangle} \\
\textbf{(def)} & E\vdash\ r.x :\text{-}\ a\ :\ \langle\!\langle R\backslash x|x{:}R.x\rangle\!\rangle
\end{array}$$

This is not quite satisfactory, because we would expect the result type to be R, meaning that the type of a record is not modified by updating one of its fields (with a value of the correct type).

Fortunately, by using **(TE9)** ($\langle\!\langle R\backslash x|x{:}R.x\rangle\!\rangle\leftrightarrow R$) we can derive the expected type rule:

(UPD)

$$\frac{E \vdash r{:}R{<:}\langle\!\langle\rangle\!\rangle \quad E \vdash a{:}R.x}{E \vdash r.x :\!\!- a \ : \ R}$$

This seems to be a compelling reason for adopting **(TE9)**, because of its impact on such an important operator as updating.

Note that the **(UPD)** rule is very strong; it applies even when *R* is a variable. From it we can derive a perhaps more natural but less general rule:

(UPD')

$$\frac{E \vdash r{:}\langle\!\langle R|x{:}A\rangle\!\rangle{<:}\langle\!\langle\rangle\!\rangle \quad E \vdash a{:}A}{E \vdash r.x :\!\!- a \ : \ \langle\!\langle R|x{:}A\rangle\!\rangle}$$

Remark. Here we might be tempted to weaken the assumption to $E \vdash a{:}A'{<:}A$, and strengthen the conclusion to $E \vdash r.x :\!\!- a : \langle\!\langle R|x{:}A'\rangle\!\rangle$. This is valid but undesirable, since we might then be unable to update the *x* field again with its original contents.

The strong **(UPD)** rule would not be expressible without *R.x* types; the following apparently natural variation is unsound:

$$\frac{E \vdash r{:}R{<:}\langle\!\langle S|x{:}A\rangle\!\rangle \quad E \vdash a{:}A}{E \vdash r.x :\!\!- a \ : \ R}$$

For example, take *A*=*Bool*, $R=\langle\!\langle x{:}True\rangle\!\rangle$, and $r=\langle x{=}true\rangle$; then from *r.x*:*Bool* and *false*:*Bool* we can derive $r.x{:}\!-\!false : \langle\!\langle x{:}True\rangle\!\rangle$.

3.12 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 show 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 $E \vdash R{<:}\langle\!\langle S|x{:}A\rangle\!\rangle$. 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 $p = x_1x_2 \dots x_k$ is a path from the root of $Tree(E \vdash R\ type)$ and $N = \{y_1, y_2, \dots, y_l\}$ is the absent set of the vertex designated by this path, then:

$$E \vdash (..(R.x_1).x_2 \dots).x_k\ type$$
$$E \vdash (..(R.x_1).x_2 \dots).x_k <: \langle\!\langle\rangle\!\rangle \backslash y_1y_2 \dots y_l$$

A convenient notational shorthand is to write $R.p$ for $(..(R.x_1).x_2 \dots).x_k$, where p is the path $p = x_1x_2 \dots x_k$. 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*.

We define $Tree(E \vdash R\ type)$ by induction on the length of E. If E has length 0 then R must be the type constant $\langle\!\langle\rangle\!\rangle$. In this case, we define:

$Tree(\emptyset \vdash \langle\!\langle\rangle\!\rangle\ type)$ = single node with empty absent set.

For context $E = E',X<:A$ we use induction on the form of type expressions:

$Tree(E \vdash Y\ type) = Tree(E' \vdash Y\ type)$ for $Y \neq X$

$Tree(E \vdash X\ type) = Tree(E' \vdash A\ type)$

$Tree(E \vdash \langle\!\langle S|x{:}B\rangle\!\rangle\ type)$ is obtained from
$T = Tree(E \vdash S\ type)$ and $T' = Tree(E \vdash B\ type)$
by making T' a subtree of the root of T along a root edge
labelled x, and removing x from the absent set of the
root of T (if there).

$Tree(E \vdash S\backslash x\ type)$ is obtained from $Tree(E \vdash S\ type)$
by deleting the subtree along the root edge labeled x (if there), and
adding x to the absent set of the root.

$Tree(E \vdash S.x\ type)$ is the subtree of $Tree(E \vdash S\ type)$
located along the root edge labeled x.

For context $E = E',X$ the definition of $Tree(E \vdash R\ type)$ is the same as above, except for the following case:

$Tree(E,X \vdash X\ type)$ = empty tree.

For context $E = E',x{:}A$ we let:

$Tree(E \vdash R\ type) = Tree(E' \vdash R\ type)$

This concludes the definition.

In the clauses defining $Tree(E \vdash \langle\!\langle S|x{:}B \rangle\!\rangle\ type)$ and $Tree(E \vdash S.x\ type)$, we have assumed certain properties of $Tree(E \vdash S\ type)$. These are justified by the following lemma.

Lemma 3.12.1:

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 $E \vdash R.p <: \langle\!\langle\rangle\!\rangle\backslash x$.

Proof

By induction on the derivation of T.

Case $\emptyset \vdash \langle\!\langle\rangle\!\rangle\ type$. Trivial.

Cases $E',X{<:}A \vdash Y\ type$ and $E',X \vdash Y\ type$ with $Y{\neq}X$.

Induction hypothesis and the property that if $E \vdash J$ for any judgment J, and $E,E'\ env$, then $E,E' \vdash J$.

Case $E',X{<:}A \vdash X\ type$.

By induction hypothesis $E' \vdash A.p\ type$ and $E' \vdash A.p <: \langle\!\langle\rangle\!\rangle\backslash x$. The conclusion follows by repeated use of **(F4)** and **(S6)**, and transitivity of <: .

Case $E',X \vdash X\ type$. Vacuous.

Cases $E',X{<:}A \vdash \langle\!\langle S|y{:}B \rangle\!\rangle\ type$ and $E',X \vdash \langle\!\langle S|y{:}B \rangle\!\rangle\ type$.

Case $p = \varepsilon$. (1) is trivial.

(2) by induction hypothesis $E \vdash S <: \langle\!\langle\rangle\!\rangle\backslash x$ for x in the absent set of T ($x{\neq}y$). Hence, $E \vdash \langle\!\langle S|y{:}B \rangle\!\rangle <: \langle\!\langle\rangle\!\rangle\backslash x$.

Case $p = yp'$. Use **(TE7)** and induction hypothesis for $E \vdash B\ type$.

Otherwise. Use **(TE8)** and induction hypothesis for $E \vdash R\ type$.

Case $E',X{<:}A \vdash S\backslash y\ type$ and $E',X \vdash S\backslash y\ type$.

Case $p = \varepsilon$. Two subcases:

Case $x{=}y$. Since $E \vdash S\backslash y\ type$ must follow from **(F3)**, we must have $E \vdash S <: \langle\!\langle\rangle\!\rangle$. The result follows by **(S4)**.

Case $x{\neq}y$. Then x must be in the absent set for $Tree(E \vdash S\ type)$ and so $E \vdash S <: \langle\!\langle\rangle\!\rangle\backslash x$. By **(S4)**, $E \vdash S\backslash y <: \langle\!\langle\rangle\!\rangle\backslash xy$, and we know that $\langle\!\langle\rangle\!\rangle\backslash xy <: \langle\!\langle\rangle\!\rangle\backslash x$.

Case $p \neq \varepsilon$. Then p must be a path in $Tree(E \vdash S\ type)$ not beginning with y. It follows from the induction hypothesis that $E \vdash S{<:} \langle\!\langle T|z{:}A \rangle\!\rangle$ for $z{\neq}y$ the first symbol of p. By **(TE4)**, we have $E \vdash S.z \leftrightarrow S\backslash y.z$, and the lemma follows by the congruence rules.

Cases $E',X{<:}A \vdash S.y\ type$ and $E',X \vdash S.y\ type$.

Straightforward from induction hypothesis.

Case $E',x{:}A \vdash R\ type$. By induction hypothesis.

□

The preceding lemma shows that the path and absent information provided by $Tree(E \vdash R\ type)$ is “sound” with respect to the proof rules of the calculus. Since the proof rules are sound with respect to our semantics, it follows that the assertions of the form $E \vdash R <: \langle\!\langle\rangle\!\rangle \backslash x$ and $\exists S, A.\ E \vdash R <: \langle\!\langle S | x{:}A \rangle\!\rangle$ determined from $Tree(E \vdash R\ type)$ are semantically sound.

We may also show that the assertions are semantically complete. It follows from the preceding lemma that the proof rules are also semantically complete for deducing assertions of the form: (1) $E \vdash R <: \langle\!\langle\rangle\!\rangle \backslash x$, and (2) if there exists S and A with $R <: \langle\!\langle S | x{:}A \rangle\!\rangle$ in every assignment satisfying E, then $E \vdash R <: \langle\!\langle S' | x{:}A' \rangle\!\rangle$ for some S' and A'.

Lemma 3.12.2:

Suppose $E \vdash R\ type$ and let $T = Tree(E \vdash R\ type)$.
There is a semantic model 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.

Proof

We may use the model constructed in section 3.8 using a single ground type $G = \mathbf{N}$, for example. For each environment E, we define an assignment ρ_E such that whenever $E \vdash R\ type$, there is some $r \in R$ with $r.p\downarrow$ iff p is a path in $Tree(E \vdash R\ type)$. (This is straightforward.) It is easy to verify that for any vertex v in any $Tree(E \vdash R\ type)$, if x is in the absent set at v, then there is no child along any edge labeled x. This and (1) imply part (2) of the lemma.

□

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 3.12.3:

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.

If $E \vdash R\ type$, then it is evident that by directing type equality rules, we may rewrite R to one of the following “normal” forms:

(1) $\langle\!\langle\rangle\!\rangle$

(2) X (a type variable)

(3) $(..(R_0.x_1) \ldots .x_i)\backslash y_1 \ldots y_j$ where R_0 is either $\langle\!\langle\rangle\!\rangle$ or a type variable.

(4) $\langle\!\langle R_0 \backslash x_1 \ldots x_i | y_1{:}A_1 \ldots y_j{:}A_j \rangle\!\rangle$ where, considering $T = Tree(E \vdash R\ type)$:

- R_0 is either $\langle\!\langle\rangle\!\rangle$ or a type variable;
- $y_1 \ldots y_j$ are exactly the labels on the root edges of T;
- $\{y_1 \ldots y_j\} \subseteq \{x_1 \ldots x_i\}$;
- $\{x_1 \ldots x_i\} - \{y_1 \ldots y_j\}$ is the absent set at the root of T;
- $A_1 \ldots A_j$ are also in normal form.

In the semantics of section 3.9, the meaning of a type expression of form (4) is a record type $R^{P,N}$, where $N=\{x_1 \ldots x_i\} - \{y_1 \ldots y_j\}$, $dom(P) = \{y_1 \ldots y_j\}$, and $P(y_n)$ is the meaning of A_n. Since we may construct models in which no type is empty, and assignments in which each type variable denotes a different type, we may show that two type expressions are provably and semantically equal iff they have the same normal forms, modulo differences in the order of field names and component types. By lemma 3.9.5, we may also see that, semantically:

$\langle\!\langle R_0 \backslash x_1 \ldots x_i | y_1{:}A_1 \ldots y_j{:}A_j \rangle\!\rangle \subseteq{:}\ \langle\!\langle S_0 \backslash u_1 \ldots u_k | v_1{:}B_1 \ldots v_l{:}B_l \rangle\!\rangle$
iff

- $(\{u_1 \ldots u_k\} - \{v_1 \ldots v_l\}) \subseteq (\{x_1 \ldots x_i\} - \{y_1 \ldots y_j\})$
- $\{v_1 \ldots v_l\} \subseteq \{y_1 \ldots y_j\}$
- if $v_m = y_n$ then $A_m \subseteq B_n$

This property allows us to decide semantic subtyping by normalizing type expressions, comparing outer-most forms, and recursively examining corresponding component types. Since all of the steps of the algorithm correspond to derivations in the proof system, completeness of the proof rules (for type equality or subtyping assertions) follows.

Theorem 3.12.4:

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. We use l, l_1, ... for sequence variables in schemas.

If S is any scheme with sequence variable l, then we say $E \vdash S\ type$ if $E \vdash S'\ type$ for every S' obtained by replacing l with a sequence of labels (including the empty sequence). If $E \vdash S\ \mathit{type}$, then a useful algorithm is *MakeAbsent*(x,S) which attempts to compute a substitution instance S' (possibly containing sequence variables) such that $E \vdash S' <: ⟪⟫\backslash x$. If such an instance exists, *MakeAbsent*(x,S) returns the smallest one. If no instance exists, the algorithm *fails*. (Algorithm *MakeAbsent* uses an extension of *Tree*($E \vdash R\ \mathit{type}$) to schemas; details are straighforward and omitted.)

Using *MakeAbsent*, we may compute a *principal type schema PTS*(E,r), for any well-formed environment E and expression r, as follows:

$$
\begin{array}{ll}
PTS(E, \langle\rangle) \ = \ ⟪⟫\backslash l & \text{(where } l \text{ is a fresh sequence variable)} \\
PTS(E, x) \ = \ E(x) & \\
PTS(E, r.x) \ = \ PTS(E, r).x \ \ \text{if defined, else fail} & \\
PTS(E, r\backslash x) \ = \ PTS(E, r)\backslash x & \\
PTS(E, \langle r|x{=}a\rangle) \ = \ ⟪MakeAbsent(x, PTS(E, r))|x{:}PTS(E, a)⟫ &
\end{array}
$$

Theorem 3.12.5:

Given $\vdash E\ \mathit{env}$ 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 $E \vdash R\ \mathit{type}$, 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. Applications and extensions

One might ask why we should go to the trouble of defining the subtle extension and restriction operators, instead of adopting the override operator as a primitive, as in [Wand 1989]. In particular, our explicit handling of negative information seems to introduce much complexity.

One answer is that negative information seems necessary to a proper understanding of the override operator. For example, the notion of *absent fields* is critical to Rémy's account of overriding in [Rémy 1989]. Hence, it seems worthwhile to investigate negative information as formalized by a separate operator.

A more pragmatic answer is that overriding really performs two different actions in different situations; it either extends a record or updates it. From a methodological point of view, a single override operator is rather undesirable because it may silently destroy information. A separate extension operator is preferable, because a type error occurs if we attempt to use it to destroy an existing field. A separate update operator is also preferable, because normally we do not want to update a field with a value of a totally different type.

Hence, in a programming language we would probably want to replace the override operator by two separate operators: one for extension, which we have, and one for updating, discussed in section 3.11. The restriction operator could still be used when we really intend to delete a field.

Admittedly, restriction is still ambiguous, because it may or may not remove a field, depending on whether the field is actually present. It is however possible to define a safe restriction operator which produces a type error if the restricted field is not present. Unfortunately, we could not find a way of completely eliminating the need for general restriction (at least on types); this operator seems necessary to express crucial well-formedness conditions.

This said, we are now ready to investigate some useful derived operators.

4.1 The override operator

The override operator $\langle r \leftarrow x{=}a\rangle =_{\text{def}} \langle r\backslash x|x{=}a\rangle$ is certainly very natural, in fact we have used it almost exclusively in our examples. The derived type rules for this operator, described below, are also very simple, especially if we consider the subsystem with only overriding and extraction. The rules mixing overriding with restriction are still rather interesting.

We recall the definition of the override operator:

$$\begin{array}{lll} \langle r \leftarrow x{=}a\rangle & =_{\text{def}} & \langle r\backslash x|x{=}a\rangle \\ \langle\!\langle R{\leftarrow}x{:}A\rangle\!\rangle & =_{\text{def}} & \langle\!\langle R\backslash x|x{:}A\rangle\!\rangle \end{array}$$

The following rules are all simply derivable from the rules for our basic operators (we assume (**TE9**)); with these, extension need not be a primitive.

Formation

$$\frac{E \vdash R{<:}\langle\!\langle\rangle\!\rangle \quad E \vdash A\ type}{E \vdash \langle\!\langle R{\leftarrow}x{:}A\rangle\!\rangle\ type} \qquad\qquad \frac{E \vdash R{<:}\langle\!\langle S{\leftarrow}x{:}A\rangle\!\rangle{<:}\langle\!\langle\rangle\!\rangle}{E \vdash R.x\ type}$$

Subtyping

$$\frac{E \vdash R{<:}\langle\!\langle\rangle\!\rangle \quad E \vdash A\ type}{E \vdash \langle\!\langle R{\leftarrow}x{:}A\rangle\!\rangle <: \langle\!\langle\rangle\!\rangle} \qquad\qquad \frac{E \vdash R{<:}S{<:}\langle\!\langle\rangle\!\rangle \quad E \vdash A{<:}B}{E \vdash \langle\!\langle R{\leftarrow}x{:}A\rangle\!\rangle <: \langle\!\langle S{\leftarrow}x{:}B\rangle\!\rangle}$$

$$\frac{E \vdash R{<:}\langle\!\langle S{\leftarrow}x{:}A\rangle\!\rangle{<:}\langle\!\langle\rangle\!\rangle}{E \vdash R.x <:\ A} \qquad\qquad \frac{E \vdash R{<:}\langle\!\langle S{\leftarrow}x{:}A\rangle\!\rangle{<:}\langle\!\langle\rangle\!\rangle}{E \vdash R <: \langle\!\langle R{\leftarrow}x{:}R.x\rangle\!\rangle}$$

Introduction/Elimination

$$\frac{E \vdash r{:}R{<:}\langle\!\langle\rangle\!\rangle \quad E \vdash a{:}A}{E \vdash \langle r{\leftarrow}x{=}a\rangle : \langle\!\langle R{\leftarrow}x{:}A\rangle\!\rangle} \qquad\qquad \frac{E \vdash r{:}\langle\!\langle R{\leftarrow}x{:}A\rangle\!\rangle{<:}\langle\!\langle\rangle\!\rangle}{E \vdash r.x : A}$$

Type Congruence

$$\frac{E \vdash R \leftrightarrow S <: \langle\!\langle\rangle\!\rangle \quad E \vdash A \leftrightarrow B}{E \vdash \langle\!\langle R{\leftarrow}x{:}A\rangle\!\rangle \leftrightarrow \langle\!\langle S{\leftarrow}x{:}B\rangle\!\rangle} \qquad\qquad \frac{E \vdash R \leftrightarrow \langle\!\langle S{\leftarrow}x{:}A\rangle\!\rangle{<:}\langle\!\langle\rangle\!\rangle}{E \vdash R.x \leftrightarrow A}$$

Type Equivalence

$$\frac{E \vdash R{<:}\langle\!\langle\rangle\!\rangle \quad E \vdash A{,}B\ type \quad x{\neq}y}{} \qquad\qquad \frac{E \vdash R{<:}\langle\!\langle S{\leftarrow}x{:}A\rangle\!\rangle{<:}\langle\!\langle\rangle\!\rangle}{}$$

$E \vdash \langle\!\langle\langle\!\langle R{\leftarrow}x{:}A\rangle\!\rangle{\leftarrow}y{:}B\rangle\!\rangle \leftrightarrow \langle\!\langle\langle\!\langle R{\leftarrow}y{:}B\rangle\!\rangle{\leftarrow}x{:}A\rangle\!\rangle$ $\qquad$ $E \vdash R \leftrightarrow \langle\!\langle R{\leftarrow}x{:}R.x\rangle\!\rangle$

$$\frac{E \vdash R{<:}\langle\!\langle\rangle\!\rangle \quad E \vdash A\ type}{E \vdash \langle\!\langle R{\leftarrow}x{:}A\rangle\!\rangle\backslash x \leftrightarrow R\backslash x} \qquad \frac{E \vdash R{<:}\langle\!\langle\rangle\!\rangle \quad E \vdash A\ type \quad x{\neq}y}{E \vdash \langle\!\langle R{\leftarrow}x{:}A\rangle\!\rangle\backslash y \leftrightarrow \langle\!\langle R\backslash y{\leftarrow}x{:}A\rangle\!\rangle}$$

$$\frac{E \vdash R{<:}\langle\!\langle\rangle\!\rangle \quad E \vdash A\ type}{E \vdash \langle\!\langle R{\leftarrow}x{:}A\rangle\!\rangle.x \leftrightarrow A} \qquad \frac{E \vdash R{<:}\langle\!\langle S{\leftarrow}y{:}B\rangle\!\rangle{<:}\langle\!\langle\rangle\!\rangle \quad E \vdash A\ type \quad x{\neq}y}{E \vdash \langle\!\langle R{\leftarrow}x{:}A\rangle\!\rangle.y \leftrightarrow R.y}$$

Value Congruence

$$\frac{E \vdash r \leftrightarrow s : R{<:}\langle\!\langle\rangle\!\rangle \quad E \vdash a \leftrightarrow b : A}{E \vdash \langle r{\leftarrow}x{=}a\rangle \leftrightarrow \langle s{\leftarrow}x{=}b\rangle : \langle\!\langle R{\leftarrow}x{:}A\rangle\!\rangle} \qquad \frac{E \vdash r \leftrightarrow s : \langle\!\langle R{\leftarrow}x{:}A\rangle\!\rangle{<:}\langle\!\langle\rangle\!\rangle}{E \vdash r.x \leftrightarrow s.x : A}$$

Value Equivalence

$$\frac{E \vdash r{:}R{<:}\langle\!\langle\rangle\!\rangle \quad E \vdash a{:}A \quad E \vdash b{:}B \quad x{\neq}y}{E \vdash \langle\langle r{\leftarrow}x{=}a\rangle{\leftarrow}y{=}b\rangle \leftrightarrow \langle\langle r{\leftarrow}y{=}b\rangle{\leftarrow}x{=}a\rangle : \langle\!\langle\langle\!\langle R{\leftarrow}x{:}A\rangle\!\rangle{\leftarrow}y{:}B\rangle\!\rangle}$$

$$\frac{E \vdash r{:}R{<:}\langle\!\langle\rangle\!\rangle \quad E \vdash a{:}A}{E \vdash \langle r{\leftarrow}x{=}a\rangle\backslash x \leftrightarrow r\backslash x : R\backslash x} \qquad \frac{E \vdash r{:}R{<:}\langle\!\langle\rangle\!\rangle \quad E \vdash a{:}A \quad x{\neq}y}{E \vdash \langle r{\leftarrow}x{=}a\rangle\backslash y \leftrightarrow \langle r\backslash y{\leftarrow}x{=}a\rangle : \langle\!\langle R{\leftarrow}x{:}A\rangle\!\rangle\backslash y}$$

$$\frac{E \vdash r{:}R{<:}\langle\!\langle\rangle\!\rangle \quad E \vdash a{:}A}{E \vdash \langle r{\leftarrow}x{=}a\rangle.x \leftrightarrow a : A} \qquad \frac{E \vdash r{:}\langle\!\langle R{\leftarrow}y{:}B\rangle\!\rangle{<:}\langle\!\langle\rangle\!\rangle \quad E \vdash a{:}A \quad x{\neq}y}{E \vdash \langle r{\leftarrow}x{=}a\rangle.y \leftrightarrow r.y : B}$$

$$\frac{E \vdash r{:}\langle\!\langle R{\leftarrow}x{:}A\rangle\!\rangle{<:}\langle\!\langle\rangle\!\rangle \quad x{\neq}y}{E \vdash r\backslash y.x \leftrightarrow r.x : A} \qquad \frac{E \vdash r{:}R{<:}\langle\!\langle S{\leftarrow}x{:}A\rangle\!\rangle{<:}\langle\!\langle\rangle\!\rangle}{E \vdash r \leftrightarrow \langle r{\leftarrow}x{=}r.x\rangle : R}$$

4.2 The rename operator

We may consider a *rename* operator, that shows another interesting use of *R.x* types.

$$r[x{\leftarrow}y] \quad =_{\text{def}} \quad \langle r\backslash x|y{=}r.x\rangle$$
$$R[x{\leftarrow}y] \quad =_{\text{def}} \quad \langle\!\langle R\backslash x|y{:}R.x\rangle\!\rangle$$

The rules for this operator are easily derived. The only interesting questions are whether renaming with an identical variable produces an equivalent value or type:

$$r[x{\leftarrow}x] \quad \leftrightarrow \quad r \qquad ?$$
$$R[x{\leftarrow}x] \quad \leftrightarrow \quad R \qquad ?$$

These equivalences are derivable for arbitrary *r* and *R*, by using:

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

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

Recall that **(VE10)** is satisfied in all our models, but **(TE9)** only holds in the latter two. These are similar to the *surjective pairing* rules in λ-calculus. An alternative, not involving surjective pairing, is to axiomatize the renaming operators independently.

4.3 The retraction operator: forgetting information

We have seen that even negative information should be considered as “additional” information. So, one might ask whether there is any way to *retract* information, both positive and negative. This would seem to be more a convenience than a necessity, since one could avoid introducing information in the first place, rather then retracting it later. However, it is still interesting to investigate the possibilities.

We have not been able to formulate operators that independently retract positive and negative information, but we can describe an operator that retracts all information about a given label in a type. This operator works purely on type information; there is no corresponding operator on values.

The *retraction* operator, $R{\sim}x$, means “forget everything about x in record type R”; the following rules enforce the cancellation of all the x information in R.

Formation/Subtyping

$$\frac{E \vdash R{<:}\langle\!\langle\rangle\!\rangle}{E \vdash R{\sim}x\ type} \qquad \frac{E \vdash R{<:}S{<:}\langle\!\langle\rangle\!\rangle}{E \vdash R{\sim}x <: S{\sim}x} \qquad \frac{E \vdash R{<:}\langle\!\langle\rangle\!\rangle}{E \vdash R <: R{\sim}x}$$

Type Equivalence

$$\frac{\vdash E\ env}{E \vdash \langle\!\langle\rangle\!\rangle{\sim}x \leftrightarrow \langle\!\langle\rangle\!\rangle} \qquad \frac{E \vdash R{<:}\langle\!\langle\rangle\!\rangle}{E \vdash R{\sim}xx \leftrightarrow R{\sim}x} \qquad \frac{E \vdash R{<:}\langle\!\langle\rangle\!\rangle}{E \vdash R{\sim}xy \leftrightarrow R{\sim}yx}$$

$$\frac{E \vdash R{<:}\langle\!\langle\rangle\!\rangle}{E \vdash R\backslash x{\sim}x \leftrightarrow R{\sim}x} \qquad \frac{E \vdash R{<:}\langle\!\langle\rangle\!\rangle \quad x{\neq}y}{E \vdash R\backslash x{\sim}y \leftrightarrow R{\sim}y\backslash x}$$

$$\frac{E \vdash R{<:}\langle\!\langle\rangle\!\rangle\backslash x \quad E \vdash A\ type}{E \vdash \langle\!\langle R|x{:}A\rangle\!\rangle{\sim}x \leftrightarrow R{\sim}x} \qquad \frac{E \vdash R{<:}\langle\!\langle\rangle\!\rangle\backslash x \quad E \vdash A\ type \quad x{\neq}y}{E \vdash \langle\!\langle R|x{:}A\rangle\!\rangle{\sim}y \leftrightarrow \langle\!\langle R{\sim}y|x{:}A\rangle\!\rangle}$$

The main consequences for values involve the rule $R <: R{\sim}x$ together with the subsumption rule: if $r{:}R$, then we are allowed to forget some information about r and conclude $r{:}R{\sim}x$.

Here are some interesting inferences:

$$\dfrac{\dfrac{E \vdash R{<:}\langle\!\langle\rangle\!\rangle}{E \vdash R{\sim}x{<:}\ \langle\!\langle\rangle\!\rangle{\sim}x}}{E \vdash R{\sim}x{<:}\ \langle\!\langle\rangle\!\rangle} \qquad \dfrac{E \vdash r{:}R \quad \dfrac{E \vdash R{<:}\langle\!\langle\rangle\!\rangle}{E \vdash R <: R{\sim}x}}{E \vdash r : R{\sim}x}$$

$$\dfrac{\dfrac{\dfrac{E \vdash r : R}{E \vdash r\backslash x : R\backslash x}}{E \vdash r\backslash x : R\backslash x{\sim}x}}{E \vdash r\backslash x : R{\sim}x} \qquad \dfrac{\dfrac{\dfrac{E \vdash r : R <: \langle\!\langle\rangle\!\rangle\backslash x \quad E \vdash a : A}{E \vdash \langle r|x{=}a\rangle : \langle\!\langle R|x{:}A\rangle\!\rangle}}{E \vdash \langle r|x{=}a\rangle : \langle\!\langle R|x{:}A\rangle\!\rangle{\sim}x}}{E \vdash \langle r|x{=}a\rangle : R{\sim}x}$$

The conclusion $r\backslash x : R{\sim}x$ above seems to say that restriction on values can be seen as a retraction operator, as well as a restriction operator.

Going back to a previous example from section 2.5, we can see the usefulness of the retraction operator for defining hierarchies in “inverse” order:

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

Note that the restriction operator would not produce the desired results.

4.4 The concatenation operator

Concatenation is a prime candidate for a primitive operator for a calculus of records. Unfortunately this operator is very difficult to handle; so difficult that we have instead chosen extension and restriction as our primitive notions. Here we discuss the main problems.

Type hierarchies are naturally expressed by a concatenation operator $R \parallel S$ on types; for example we would like to define:

$let\ ColorDisc = ColorPoint \parallel Disc$

Given a corresponding operator of values, $r \parallel s$ of type $R \parallel S$ for $r{:}R$ and $s{:}S$, we would like to guarantee that if we can derive $r \parallel s : R \parallel S$ then there is a succesful and unambiguous way to execute $r \parallel s$ at run-time.

Under these conditions, we can see that concatenation is in fundamental conflict with the subsumption rule. Consider the function:

$let\ f1(X{<:}\langle\!\langle x{:}Int \rangle\!\rangle)(Y{<:}\langle\!\langle y{:}Bool \rangle\!\rangle)(r{:}X)(s{:}Y) : X \parallel Y = r \parallel s$
$f1(\langle\!\langle x{:}Int,\ z{:}Int \rangle\!\rangle)(\langle\!\langle y{:}Bool,\ z{:}Bool \rangle\!\rangle)(\langle x{=}3,\ z{=}4 \rangle)(\langle x{=}3,\ z{=}true \rangle) \ \leftrightarrow\ \ ?\ :\ ?$

There is no explicit conflict in the definition of $f1$, so it should typecheck. But when $f1$ is used as above, we have to decide which z field to produce, both in the result type and in the result value. A popular choice is to have $X \parallel Y$ perform a left-to-right (or right-to-left) overriding of common fields; similarly for $r \parallel s$ at run-time. However, run-time overriding can run into difficulties:

$let\ f2(r{:}\langle\!\langle x{:}Int \rangle\!\rangle)(s{:}\langle\!\langle y{:}Bool \rangle\!\rangle) : \langle\!\langle x{:}Int,\ y{:}Bool \rangle\!\rangle = r \parallel s$
$f2(\langle x{=}3,\ y{=}4 \rangle)(\langle y{=}true,\ x{=}false \rangle) \ \leftrightarrow\ \ ?$

Let us assume here that, whatever definition we give to $\parallel$, it satisfies the equation: $\langle\!\langle x{:}Int \rangle\!\rangle \parallel \langle\!\langle y{:}Bool \rangle\!\rangle = \langle\!\langle x{:}Int,\ y{:}Bool \rangle\!\rangle$; then $f2$ is well-typed. Could we use run-time overriding in the invocation of $f2$ above? According to the result type of $f2$, the left x should override the right x, while the right y should override the left y, so monodirectional overriding will not work.

An option here is to give a run-time error, but this seems to defeat the purpose of typechecking $r \parallel s$. Another option might be to compile special code for $r \parallel s$, according to the types of r and s, so as to pick the x field from r and the y field from s, and to do overriding on the additional fields (to deal with the polymorphic case, below). This idea however runs into further difficulties:

$$f1(\langle\!\langle x{:}Int,\, y{:}Int,\, z{:}Int\rangle\!\rangle)(\langle\!\langle y{:}Bool,\, x{:}Bool,\, z{:}Bool\rangle\!\rangle)$$
$$(\langle x{=}3,\, y{=}4,\, z{=}4\rangle)(\langle y{=}true,\, x{=}false,\, z{=}true\rangle) \;\leftrightarrow\; ? \;:\; ?$$

If $X \parallel Y$ is computed by overriding, here, we get the wrong result. Making $X \parallel Y$ compatible with the behavior of $r \parallel s$ above, would require violating some basic rules, such as the beta-conversion rules for type parameters.

Because of all these difficulties, we should now feel compelled to define $R \parallel S$ only when R and S are disjoint: that is when any field present in an element of R is absent from every element of S, and vice versa. Unfortunately, there is no way to axiomatize this notion without drastically changing our type system: any two record types R and S have a non-empty intersection, and an element of this intersection can be exhibited via the subsumption rule.

5. 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.

Appendix

This appendix contains soundness proofs for the semantic interpretations given in the paper.

Semantics of the pure calculus of records

System *S1* consists 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 systems *S1* are sound with respect to the interpretation of judgments given in section 3.8.

Proof

By induction on the length of the derivation of the judgments.

Environments

(ENV1). Vacuously true.

(ENV2). Vacuously true.

(ENV3). By hypothesis, $E \vdash A$ *type* and so $A_\rho \subseteq V$ for any ρ satisfying E. Moreover, E is well-formed by lemma 3.7.1, hence $E,X<:A$ is also well-formed.

(ENV4). Similar to **(ENV3)**.

Variables

(VAR1). If ρ satisfies E,X,E', then by definition $\rho(X) \subseteq V$.

(VAR2). If $\vdash E,X<:A,E'$ *env*, then for any ρ satisfying E we have $A_\rho \subseteq V$. Thus any ρ satisfying $E,X<:A,E'$ must yield $\rho(X) \subseteq A_\rho \subseteq V$.

(VAR3). Similar to **(VAR2)**.

(VAR4). If $\vdash E,x:A,E'$ *env*, then for any ρ satisfying E we have $\rho(x) \in A_\rho \subseteq V$. Thus any ρ satisfying $E,x:A,E'$ must yield $\rho(x) \in A_\rho \subseteq V$.

General

(G1). If, for every ρ satisfying E, $A_\rho=B_\rho \subseteq V$ then $A_\rho \subseteq B_\rho$.

(G2). By transitivity of subset.

(G3). If, for every ρ satisfying E, $a_\rho \in A_\rho$ and $A_\rho \subseteq B_\rho$ then $a_\rho \in B_\rho$.

(G4). By symmetry of equality.

(G5). By transitivity of equality.

(G6). If, for every ρ satisfying E, $a_\rho=b_\rho \in A_\rho$ then $b_\rho=a_\rho \in A_\rho$.

(G7). If, for every ρ satisfying E, $a_\rho=b_\rho \in A_\rho$ and $b_\rho=c_\rho \in A_\rho$ then $a_\rho=c_\rho \in A_\rho$.

Formation

(F1). $R \subseteq V$

(F2). If, for every ρ satisfying E, $R_\rho \subseteq R\ \text{-}x$ and $A_\rho \subseteq V$ then $R_\rho[x:A_\rho] \subseteq R \subseteq V$, by Lemma 3.8.2.

(F3). If $R_\rho \subseteq R$, then $R_\rho\text{-}x \subseteq R \subseteq V$, by Lemma 3.8.2.

(F4). If $R_\rho \subseteq S_\rho[x:A_\rho] \subseteq R$, then $A_\rho \subseteq V$; hence $R_\rho(x) \subseteq V$ by Lemma 3.8.2.

Subtyping

(S1). If, for every ρ satisfying E, $R_\rho \subseteq R$ -x, then R_ρ is a set of finite functions $r \in L \rightharpoonup_{\text{fin}} V$ with $x \notin dom(r)$. For each such r, and any $a \in A_\rho \subseteq V$, we have $r[x{=}a] \in L \rightharpoonup_{\text{fin}} V$. Thus $R_\rho[x{:}A_\rho] \subseteq R$.

(S2). If $R_\rho \subseteq R$, then $R_\rho\text{-}x \subseteq R_\rho \subseteq R$.

(S3). Suppose $R_\rho \subseteq S_\rho \subseteq R$ -x and $A_\rho \subseteq B_\rho \subseteq V$. Let $r\in R_\rho[x{:}A_\rho]$. This means $\exists s\in R_\rho$ with $r = s[x{=}a]$. Since $s\in S_\rho$ and $A_\rho \subseteq B_\rho$, we have $s[x{=}a] \in S_\rho[x{:}B_\rho]$. Hence $R_\rho[x{:}A_\rho] \subseteq S_\rho[x{:}B_\rho]$.

(S4). Suppose $R_\rho \subseteq S_\rho \subseteq R$. If $r'\in R_\rho\text{-}x$, then $r' = r\text{-}x$ for some $r\in R_\rho$. Since $r\in S_\rho$, it follows that $r' = r\text{-}x \in S_\rho\text{-}x$.

(S5). Suppose $R_\rho \subseteq S_\rho[x{:}A_\rho] \subseteq R$, then for any $r\in R_\rho$, $r\in S_\rho[x{:}A_\rho] = \{s[x{=}a] \mid s\in S_\rho,\ a\in A_\rho\}$; so that $r(x)\in A_\rho$. Hence $R_\rho(x) = \{r(x) \mid r\in R_\rho\} \subseteq A_\rho$.

(S6). Suppose $R_\rho \subseteq S_\rho[x{:}A_\rho] \subseteq R$, then for any $r\in R_\rho$, $r\in S_\rho[x{:}A]$, so that $r{=}s[x{=}a]$ for some $s\in S_\rho$ and $a\in A_\rho$. We have $a{=}r(x)\in R_\rho(x)$, and $s{=}r\text{-}x\in R_\rho\text{-}x$, hence $r{=}(r\text{-}x)[x{=}r(x)]\in(R_\rho\text{-}x)[x{:}R_\rho(x)]$. It follows that $R_\rho\subseteq(R_\rho\text{-}x)[x{:}R_\rho(x)]$.

Introduction

(I1). $\emptyset \in R \subseteq V$.

(I2). If, for every ρ satisfying E, the empty function $\emptyset \in R_\rho \subseteq R$, then $\emptyset = \emptyset\ \text{-}x_1..x_n \in R_\rho\text{-}y \subseteq R$.

(I3). If $r_\rho\in R_\rho$ with $x \notin dom(r_\rho)$ and $a_\rho\in A_\rho$, then $r_\rho[x{=}a_\rho]$ is well-defined, by Lemma 3.8.1, and belongs to $R_\rho[x{:}A_\rho] \subseteq R$, by Lemma 3.8.2.

Elimination

(E1). If, for every ρ satisfying E, $r_\rho\in R_\rho \subseteq R$, then $x \notin dom(r_\rho\text{-}x)$. Hence $r_\rho\text{-}x \in R_\rho\text{-}x \subseteq R$, by Lemma 3.8.2.

(E2). If $r_\rho\in R_\rho[x{:}A_\rho] \subseteq R$, then $A_\rho \subseteq V$, and r_ρ is a record with $r_\rho(x)\in A_\rho$.

Type congruence

(TC1). $R = R \subseteq V$.

(TC2). For every ρ satisfying E, $X_\rho = X_\rho \subseteq V$.

(TC3). Suppose $R_\rho{=}S_\rho$, $S_\rho \subseteq R$ -x, and $A_\rho{=}B_\rho \subseteq V$. Then $R_\rho[x{:}A_\rho] = S_\rho[x{:}B_\rho] \subseteq R \subseteq V$.

(TC4). Suppose $R_\rho{=}S_\rho \subseteq R$, then $R_\rho\text{-}x{=}S_\rho\text{-}x \subseteq R \subseteq V$.

(TC5). Suppose $R_\rho{=}S_\rho \subseteq T_\rho[x{:}A_\rho] \subseteq R$. Then both R_ρ and S_ρ are sets of functions r with $x \notin dom(r)$. Hence $R_\rho(x) = \{r(x) \mid r\in R_\rho\} = \{r(x) \mid r\in S_\rho\} = S_\rho(x) \subseteq V$.

Type equivalence

(TE1). Suppose, for every ρ satisfying E, $R_\rho \subseteq (R\ \text{-}x)\text{-}y$, $A_\rho,B_\rho \subseteq V$, and $x,y \in L$. For each $r\in R_\rho$, $x,y \notin dom(r)$. Then,

$R_\rho[x{:}A_\rho][y{:}B_\rho] =$

$\{s[y{=}b] \mid s\in\{r[x{=}a] \mid r\in R_\rho,\ a\in A_\rho\},\ b\in B_\rho\} =$

$\{r[x{=}a][y{=}b] \mid r\in R_\rho, a\in A_\rho, b\in B_\rho\} = \{r[y{=}b][x{=}a] \mid r\in R_\rho, b\in B_\rho, a\in A_\rho\}=$

$\{s[x{=}a] \mid s \in \{r[y{=}b] \mid r \in R_\rho, b \in B_\rho\}, a \in A_\rho\} =$
$R_\rho[y{:}B_\rho][x{:}A_\rho] \subseteq R \subseteq V.$

(TE2). If $R_\rho \subseteq R$ -x, then R_ρ is a set of r with $x \notin dom(r)$. Hence R_ρ-$x = R_\rho$.

(TE3). If $R_\rho \subseteq R$ then $(R_\rho\text{-}x)\text{-}y = (R_\rho\text{-}y)\text{-}x$.

(TE4). Suppose $R_\rho \subseteq S_\rho[y{:}B_\rho] \subseteq R$ and $x \neq y$.
For each $r \in R_\rho$, $y \notin dom(r)$. Then,
$(R_\rho\text{-}x)(y) =$
$\{s(y) \mid s \in \{r\text{-}x \mid r \in R_\rho\}\} = \{(r\text{-}x)(y) \mid r \in R_\rho\} = \{r(y) \mid r \in R_\rho\} =$
$R_\rho(y) \subseteq V.$

(TE5). Suppose $R_\rho \subseteq R$ -x and $A_\rho \subseteq V$.
Then $R_\rho[x{:}A_\rho] = \{r[x{=}a] \mid r \in R_\rho, a \in A_\rho\}$.
So $(R_\rho[x{:}A_\rho])\text{-}x = \{r \mid r \in R_\rho\} = R_\rho$.

(TE6). Suppose $R_\rho \subseteq R$ -x, $A_\rho \subseteq V$, and $x \neq y$. Then,
$(R_\rho[x{:}A_\rho])\text{-}y =$
$\{(r[x{=}a])\text{-}y \mid r \in R_\rho, a \in A_\rho\} = \{(r\text{-}y)[x{=}a] \mid r \in R_\rho, a \in A_\rho\} =$
$(R_\rho\text{-}y)[x{:}A_\rho] \subseteq R \subseteq V.$

(TE7). Suppose $R_\rho \subseteq R$ -x and $A_\rho \subseteq V$.
Then $R_\rho[x{:}A_\rho] = \{r[x{=}a] \mid r \in R_\rho, a \in A_\rho\}$.
Hence $(R_\rho[x{:}A_\rho])(x) = \{(r[x{=}a])(x) \mid r \in R_\rho, a \in A_\rho\} = A_\rho \subseteq V$.

(TE8). Suppose $R_\rho \subseteq S_\rho[y{:}B_\rho]\text{-}x \subseteq R$, $A_\rho \subseteq V$, and $x \neq y$. Then,
$(R_\rho[x{:}A_\rho])(y) = \{(r[x{=}a])(y) \mid r \in R_\rho, a \in A_\rho\} = \{r(y) \mid r \in R_\rho\} = R_\rho(y) \subseteq V.$

Value congruence

(TC1). $\emptyset = \emptyset \subseteq R$

(TC2). If, for every ρ satisfying E, $\rho(x) \in A_\rho \subseteq V$, then $\rho(x) = \rho(x) \in A_\rho$.

(TC3). Suppose $r_\rho = s_\rho \in R_\rho \subseteq R$ -x and $a_\rho = b_\rho \in A_\rho \subseteq V$.
Then $x \notin dom(r_\rho) \cup dom(s_\rho)$. Hence $r_\rho[x{=}a_\rho] = s_\rho[x{=}b_\rho] \in R_\rho[x{:}A_\rho] \subseteq R$,
by case **(I3)**.

(TC4). Suppose $r_\rho = s_\rho \in R_\rho \subseteq R$. Then $r_\rho\text{-}x = s_\rho\text{-}x \in R_\rho\text{-}x \subseteq R$, by case **(E1)**.

(TC5). Suppose $r_\rho = s_\rho \in R_\rho \subseteq S_\rho[x{:}A_\rho] \subseteq R$.
Then $R_\rho \subseteq (R_\rho\text{-}x)[x{:}R_\rho(x)]$ (by case **(S6)**), and $r_\rho, s_\rho \in (R_\rho\text{-}x)[x{:}R_\rho(x)]$.
Hence, by case **(E2)**, $r_\rho(x) = s_\rho(x) \in R_\rho(x) \subseteq V$.

Value equivalence

(VE1). Suppose, for every ρ satisfying E, $r_\rho \in R_\rho \subseteq R$ -x-y, $a_\rho \in A_\rho \subseteq V$,
$b_\rho \in B_\rho \subseteq V$, and $x \neq y$. Then, $x,y \notin dom(r_\rho)$, and
$r_\rho[x{=}a_\rho][y{=}b_\rho] = r_\rho[y{=}b_\rho][x{=}a_\rho] \in R_\rho[x{:}A_\rho][y{:}B_\rho] \subseteq R$.

(VE2). $\emptyset$ -$x = \emptyset \in R$.

(VE3). Suppose $r_\rho \in R_\rho \subseteq R$ -x. Since $x \notin dom(r_\rho)$, $r_\rho\text{-}x = r_\rho$.

(VE4). Suppose $r_\rho \in R_\rho \subseteq R$. $(r_\rho\text{-}x)\text{-}y = (r_\rho\text{-}y)\text{-}x \in (R_\rho\text{-}x)\text{-}y \subseteq R$.

(VE5). Suppose $r_\rho \in R_\rho[x{:}A_\rho] \subseteq R$ and $x \neq y$.
Then $x \in dom(r_\rho)$ and $r_\rho\text{-}y.x = r_\rho.x \in A_\rho \subseteq V$.

(VE6). Suppose $r_\rho \in R_\rho \subseteq R$ -x and $a_\rho \in A_\rho \subseteq V$.
Then $x \notin dom(r_\rho)$ and $r_\rho[x{=}a_\rho]\text{-}x = r_\rho$.

(VE7). Suppose $r_\rho \in R_\rho \subseteq R$ -x, $a_\rho \in A_\rho \subseteq V$ and $x \neq y$.

Then $x \notin dom(r_\rho)$ and $(r_\rho[x{=}a_\rho])\text{-}y = (r_\rho\text{-}y)[x{=}a_\rho] \in (R_\rho[x{:}A_\rho])\text{-}y \subseteq R$.

(VE8). Suppose $r_\rho \in R_\rho \subseteq R\ \text{-}x$, and $a_\rho \in A_\rho \subseteq V$.

Then $x \notin dom(r_\rho)$ and $(r_\rho[x{=}a_\rho])(x) = a_\rho$.

(VE9). Suppose $r_\rho \in R_\rho[y{:}B_\rho]\text{-}x \subseteq R$, $a_\rho \in A_\rho \subseteq V$ and $x \neq y$.

Then $B_\rho \subseteq V$, $x \notin dom(r_\rho)$, $y \in dom(r_\rho)$, and $(r_\rho[x{=}a_\rho])(y) = r_\rho(y) \in B_\rho$.

(VE10). Suppose $r_\rho \in R_\rho \subseteq S_\rho[x{:}A_\rho] \subseteq R$.

Then $r_\rho \in S_\rho[x{:}A_\rho]$, so that $r_\rho{=}s[x{=}a]$ for some $s \in S_\rho$ and $a \in A_\rho$.
We have $a{=}r_\rho(x) \in R_\rho(x)$, and $s{=}r_\rho\text{-}x \in R_\rho\text{-}x$, hence $r_\rho{=}(r_\rho\text{-}x)[x{=}r_\rho(x)]$,
which is well-formed (is a member of $(R_\rho\text{-}x)[x{:}R_\rho(x)]$).

□

A construction giving *R* = 《*R*\x|*x*:*R*.*x*》

System *S2* is system *S1* of Theorem 3.8.3 plus the rule **(TE9)**.

Theorem 3.9.7 (soundness):

The inference rules of system *S2* are sound with respect to the interpretation of judgments given in section 3.9.

Proof

The proof follows the general pattern of Theorem 3.8.3. The main new properties that are needed are proved as lemmas in section 3.9.
In particular, **(TE9)** follows from Lemma 3.9.6. The formation rules come from Lemmas 3.9.2, 3.9.3, 3.9.4, and 3.9.5.

□

An extensional model construction

System *S3* is system *S1* of Theorem 3.8.3 plus the rules **(TE9)** and **(VC1b)**.

Theorem 3.10.4 (soundness):

The inference rules of system *S3* are sound for the PER model construction given in section 3.10.

Proof

The proof follows the general pattern of Theorem 3.8.3, using the lemmas proved in section 3.10

□

References

[Breazu-Tannen, *et al.* 1989] V. Breazu-Tannen, T. Coquand, C. Gunter, and A. Scedrov. **Inheritance and explicit coercion**. *Proc. 4th Annual IEEE Symposium on Logic in Computer Science*.

[Bruce, Longo 1990] K.B. Bruce and G. Longo, **A modest model of records, inheritance and bounded quantification, Information and Computation**. *Information and Computation* **87**(1/2), 196-240.

[Bruce, Meyer, Mitchell 1990] K.B. Bruce, A.R. Meyer, and J.C. Mitchell, **The semantics of second order lambda calculus**. *Information and Computation* **85**(1), 76-134.

[Cardelli 1988] L. Cardelli, **A semantics of multiple inheritance**. *Information and Computation* **76**, 138-164.

[Cardelli, *et al.* 1989] L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, and G. Nelson. **Modula-3 report (revised)**. Research Report n.52. DEC Systems Research Center.

[Cardelli, Wegner 1985] L. Cardelli and P. Wegner, **On understanding types, data abstraction and polymorphism**. *Computing Surveys* **17**(4), 471-522.

[Curien, Ghelli 1992] P.-L. Curien and G. Ghelli, **Coherence of subsumption, minimum typing and type-checking in $F_{\leq}$**. *Mathematical Structures in Computer Science* **2**(1), 55-91.

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

[Girard 1971] 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**. *Proc. Second Scandinavian Logic Symposium*. North-Holland.

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

[Jategaonkar, Mitchell 1988] L.A. Jategaonkar and J.C. Mitchell. **ML with extended pattern matching and subtypes**. ACM Conference on Lisp and Functional Programming.

[Longo, Moggi 1991] G. Longo and E. Moggi, **Constructive natural deduction and its 'ω-set' interpretation**. *Mathematical Structures in Computer Science* **1**(2).

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

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

[Mitchell 1984] J.C. Mitchell. **Coercion and type inference**. *Proc. 11th Annual ACM Symposium on Principles of Programming Languages*.

[Mitchell 1986] J.C. Mitchell. **A type inference approach to reduction properties and semantics of polymorphic expressions (summary)**. *Proc. Symposium on Lisp and Functional Programming*.

[Mitchell 1990] J.C. Mitchell, **Type systems for programming languages**. In *Handbook of Theoretical Computer Science,* J. van Leeuwen, ed. North Holland. 365-458.

[Ohori, Buneman 1988] A. Ohori and P. Buneman. **Type inference in a database programming language**. *Proc. ACM Conference on LISP and Functional Programming*.

[Ohori, Buneman, Breazu-Tannen 1988] A. Ohori, P. Buneman, and 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.

[Rémy 1989] D. Rémy. **Typechecking records and variants in a natural extension of ML**. *Proc. 16th Annual ACM Symposium on Principles of Programming Languages*.

[Reynolds 1974] J.C. Reynolds. **Towards a theory of type structure**. *Proc. Colloquium sur la programmation*. Lecture Notes in Computer Science 19. Springer-Verlag.

[Schaffert, *et al.* 1986] C. Schaffert, T. Cooper, B. Bullis, M. Kilian, and C. Wilpolt. **An introduction to Trellis/Owl**. *Proc. ACM Conference on Object Oriented Programming Systems, Languages, and Applications*.

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

[Wand 1987] M. Wand. **Complete type inference for simple objects**. *Proc. 2nd Annual IEEE Symposium on Logic in Computer Science*.

[Wand 1989] M. Wand. **Type inference for record concatenation and multiple inheritance**. *Proc. 4th Annual IEEE Symposium on Logic in Computer Science*.