Computing with Taxonomies

Luca Cardelli

Bell Laboratories
Murray Hill, New Jersey 07974

It next will be right
To describe each particular batch:
Distinguishing those that have feathers, and bite,
From those that have whiskers, and scratch.
(Lewis Carroll)

1. Introduction

The notions of inheritance and object-oriented programming first appeared in Simula 67 [Dahl 66]. In Simula, objects can be grouped into classes and classes can be organized into a subclass hierarchy. Objects are similar to records with functional components, and elements of a class can appear wherever elements of the respective superclasses are expected. Subclasses inherit all the attributes of their superclasses. In Simula, the issues are somewhat complicated by the use of objects as coroutines, so that communication between objects can be interpreted as "message-passing" between processes.

Smalltalk [Goldberg 83] adopts and exploits the idea of inheritance, with some changes. While stressing the message-passing paradigm, Smalltalk does not have coroutines. Message passing is just function call, although the association of message names to functions (called methods) is not straightforward. With respect to Simula, Smalltalk also abandons static scoping, to gain flexibility in interactive use, and strong typing, allowing it to implement system introspection and to introduce the notion of meta-classes.

Inheritance can be single or multiple. In the case of single inheritance, as in Simula or Smalltalk, the subclass hierarchy has the form of a tree, i.e. every class has a unique superclass. Examples can be shown where a class can be indifferently considered a subclass of two incompatible superclasses; then an arbitrary decision has to be made to determine which superclass to use. This problem leads naturally to the idea of multiple inheritance.

Multiple inheritance occurs when an object can belong to several incomparable superclasses: the subclass relation is no longer constrained to form a tree, but can form a dag. Multiple inheritance is more elegant than simple inheritance, but more difficult to implement. So far, it has mostly been considered in the context of type-free dynamically-scoped languages and implemented as Lisp or Smalltalk extensions [Weinreb 81, Steels 83], or as part of knowledge representation languages [Attardi 81].

The differences between Simula, Smalltalk and other languages suggest that inheritance is the only notion critically associated with object-oriented programming. Coroutines, message-passing, static/dynamic scoping, typechecking and single/multiple superclasses are all fairly independent issues. Hence, a theory of object-oriented programming should first of all focus on the meaning of inheritance.

The concept of inheritance has been used extensively in artificial intelligence to express taxonomies of objects. It is also advocated as a natural way of organizing data schemas in database applications [Borgida 82]. The motivation of this work is to estabilish formal foundations for the

Galileo database language [Albano 83] which uses multiple inheritance as a major data-structuring facility.

The scope of this paper is to present a clean semantics of multiple inheritance in the context of strongly-typed, statically-scoped languages. A sound typechecking algorithm is described in a forthcoming paper. Multiple inheritance is interpreted in a broad sense: instead of being limited to objects, it is extended in a natural way to union types and to higher functional types.

2. Objects as Records

There are several ways of thinking of what objects *are*. In the pure Smalltalk-like view, objects recall physical entities, like boxes or cars. Physical entities are unfortunately not very useful as semantic models of objects, because they are far too complicated to describe formally.

Two simpler interpretations of objects seem to emerge from the implementations of object-oriented languages. The first interpretation derives from Simula, where objects are essentially records with possibly functional components. Message passing is field selection and inheritance has to do with the number and type of fields possessed by a record.

The second interpretation derives from Lisp. An object is a function which receives a message (a string or an atom) and dispatches on the message to select the appropriate "method". Here message-passing is function application and inheritance has to do with the way messages are dispatched.

In some sense these two interpretations are equivalent because records can be represented as functions from labels (messages) to values. However, to say that objects are functions is misleading, because we must qualify that objects are functions over messages. Instead we can safely assert that objects are records, because labels are an essential part of records.

We also want to regard objects as records for typechecking purposes. While a (character string) message can be the result of an arbitrary computation, a record selection usually requires the selection label to be known at compile-time. In the latter case it is possible to statically determine the set of messages supported by an object, and a compile-time type error can be reported on any attempt to send unsupported messages. This property is true for Simula, but has been lost in all the succeeding languages.

We shall show how records can account for all the basic features of objects, provided that the surrounding language is rich enough. The features we consider are multiple inheritance, message-passing, private instance variables and the concept of "self". The duality between records and functions however remains: in our language objects are records, but in the semantics records are functions.

3. Records

A **record** is a finite association of values to labels, for example:

$$(a = 3, b = true, c = "abc")$$

is a record with three fields a, b and c having as values an integer 3, a boolean *true* and a string *"abc"* respectively. The **labels** a, b and c belong to a separate domain of labels; they are not identifiers or strings, and cannot be computed as the result of expressions. Records are unordered and cannot contain the same label twice.

The basic operation on records is field selection, denoted by the usual **dot** notation:

$$(a = 3, b = true, c = "abc") \,.\, a \equiv 3$$

An expression can have one or more types; we write

$$e : \tau$$

to indicate that expression e has type τ.

Records have **record types** which are labeled sets of types with distinct labels, for example

we have:

$(a = 3, b = true) : (a : int, b : bool)$

In general, we can write the following informal typing rule for records:

[Rule1] if $e_1: \tau_1$ and .. and $e_n: \tau_n$ then $(a_1 = e_1, .. , a_n = e_n) : (a_1: \tau_1, .. , a_n: \tau_n)$

This is the first of a series of informal rules which are only meant to capture our initial intuitions about typing. They are not supposed to form a complete set or to be independent of each other.

There is a **subtype** relation on record types which corresponds to the *subclass* relation of Simula and Smalltalk. For example we may define the following types:

type any = ()
type object = (*age*: *int*)
type vehicle = (*age*: *int*, *speed*: *int*)
type machine = (*age*: *int*, *fuel*: *string*)
type car = (*age*: *int*, *speed*: *int*, *fuel*: *string*)

Intuitively a vehicle *is* an object, a machine *is* an object and a car *is* a vehicle *and* a machine (and therefore an object). We say that *car* is a subtype of *machine* and *vehicle*; *machine* is a subtype of *object*; etc. In general a record type τ is a subtype (written $\leq$) of a record type τ' if τ has all the fields of τ', and possibly more, and the common fields of τ and τ' are in the $\leq$ relation. Moreover, all the basic types (like *int* and *bool*) are subtypes of themselves:

[Rule2]
- $\iota \leq \iota$ (ι a basic type)
- $\tau_1 \leq \tau'_1 \ldots \tau_n \leq \tau'_n \implies (a_1: \tau_1, \ldots, a_{n+m}: \tau_{n+m}) \leq (a_1: \tau'_1, \ldots, a_n: \tau'_n)$

Let us consider a particular car (value definitions are prefixed by the keyword *val*):

val mycar = (*age* = 4, *speed* = 140, *fuel* = "*gasoline*")

Of course *mycar*: *car* (*mycar* has type *car*), but we might also want to assert *mycar*: *object*. To obtain this, we say that when a value has a type τ, then it has also all the types τ' such that τ is a subtype of τ'. This leads to our third informal type rule:

[Rule3] if $a : \tau$ and $\tau \leq \tau'$ then $a : \tau'$

If we define the function:

val age(*x*: *object*): *int* = *x.age*

we can meaningfully compute *age*(*mycar*) as, by [Rule3] *mycar* has the type required by *age*. Indeed *mycar* has the types *car*, *vehicle*, *machine*, *object*, the empty record type and many other ones.

When is it meaningful to apply a function to an argument? This is determined by the following rules:

[Rule4] if $f: \sigma \to \tau$ and $a: \sigma$ then $f(a)$ is meaningful, and $f(a): \tau$

[Rule5] if $f: \sigma \to \tau$ and $a: \sigma'$, where $\sigma' \leq \sigma$ then $f(a)$ is meaningful, and $f(a): \tau$

[Rule5] is just a consequence of [Rule3] and [Rule4]. From [Rule3] we can deduce that $a : \sigma$; than it is certainly meaningful to compute $f(a)$ as $f: \sigma \to \tau$.

The conventional *subclass* relation is usually only defined on objects or classes. Our *subtype* relation also extends naturally to functional types. Consider the function

serial_number: *int* → *car*

We can argue that *serial_number* returns vehicles, as all cars are vehicles. In general, all *car*-

valued function are also *vehicle*-valued functions, so that for any domain type t we can say that $t \to car$ (an appropriate domain of functions from t to *car*) is a subtype of $t \to vehicle$:

$$t \to car \ \leq \ t \to vehicle \quad \text{because} \quad car \ \leq \ vehicle$$

Now consider the function:

$$speed: vehicle \to int$$

As all cars are vehicles, we can use this function to compute the speed of a car. Hence *speed* is also a function from *car* to integer. In general every function on vehicles is also a function on cars, and we can say that $vehicle \to int$ is a subtype of $car \to int$:

$$vehicle \to t \ \leq \ car \to t \quad \text{because} \quad car \ \leq \ vehicle$$

Something interesting is happening here: note how the subtype relation is inverted on the left hand side of the arrow. This happens because of the particular meaning we are giving to the $\to$ operator, as explained formally in the following sections. We are assuming a universal value domain V of all computable values. Every function f is a function from V to V, written $f: V -> V$, where "$->$" is the conventional continuous function space. By $f: \sigma \to \tau$ we indicate a function $f: V -> V$ which whenever given an element of $\sigma \subseteq V$ returns an element of $\tau \subseteq V$ (nothing is asserted about the behavior of f outside σ).

Given any function $f: \sigma \to \tau$ from some domain σ to some codomain τ, we can always consider it as a function from some smaller domain $\sigma' \subseteq \sigma$ to some bigger codomain $\tau' \supseteq \tau$. For example a function f: *vehicle* $\to$ *vehicle* can be used in the context $age(f(mycar))$, where it is used as a function f: *car* $\to$ *object* (the application $f(mycar)$ makes sense because every car is a vehicle; $v=f(mycar)$ is a vehicle; hence it makes sense to compute $age(v)$ as every vehicle is an object).

The general rule of subtyping among functional types can be expressed as follows:

[Rule6] if $\sigma' \leq \sigma$ and $\tau \leq \tau'$ then $\sigma \to \tau \ \leq \ \sigma' \to \tau'$

As we said, the subtype relation extends to higher types. For example, the following is a definition of a function *mycar_attribute* which takes any integer-valued function on cars and applies it to my car.

$$val\ mycar\_attribute(f: car \to int): int = f(mycar)$$

We can then apply it to functions of any type which is a subtype of $car \to int$, e.g., $age: object \to int$. (Why? Because *car* is a subtype of *object*, hence $object \to int$ is a subtype of $car \to int$, [Rule6] hence $(mycar\_attribute: (car \to int) \to int)(age: object \to int)$ makes sense [Rule5]).

$$mycar\_attribute(age) \equiv 4$$

$$mycar\_attribute(speed) \equiv 140$$

Up to now we proceeded by assigning certain types to certain values. However the subtype relation has a very strong intuitive flavor of **inclusion** of types considered as sets of objects, and we want justify our type assignments on semantic grounds.

Semantically we could regard the type *vehicle* as the set of all the records with a field *age* and a field *speed* having the appropriate types, but then cars would not belong to the set of vehicles as they have three fields while vehicles have two. To obtain the inclusion that we intuitively expect, we must say that the type vehicle is the set of all records which have *at least* two fields as above, but may have other fields. In this sense a car is a vehicle, and the set of all cars is included in the set of all vehicles, as we might expect. Some care is however needed to define these "sets", and this will be done formally in the following sections.

Record types can have a large number of fields, hence we need some notation for quickly defining a subtype of some record type, without having to list again all the fields of the record

type. The following three sets of definitions are equivalent:

type object = (*age*: *int*)
type vehicle = (*age*: *int*, *speed*: *int*)
type machine = (*age*: *int*, *fuel*: *string*)
type car = (*age*: *int*, *speed*: *int*, *fuel*: *string*)

type object = (*age*: *int*)
type vehicle = *object and* (*speed*: *int*)
type machine = *object and* (*fuel*: *string*)
type car = *vehicle and machine*

type object = (*age*: *int*)
type car = *object and* (*speed*: *int*, *fuel*: *string*)
type vehicle = *car ignoring fuel*
type machine = *car ignoring speed*

The *and* operator forms the union of the fields of two record types; if two record types have some labels in common (like in *vehicle and machine*), then the corresponding types must match. At this point we do not specify exactly what "match" means, except that in the example above "matching" is equivalent to "being the same". In its full generality, *and* corresponds to a *join* operation on type expressions, as explained in a later section.

The *ignoring* operator simply eliminates a component from a record type; it is undefined on other types.

4. Variants

The two basic non-functional data type constructions in denotational semantics are cartesian products and disjoint sums. We have seen that inheritance can be expressed as a subtype relation on record types, which then extends to higher types. Record types are just labeled cartesian products, and by analogy we can ask whether there is some similar notion deriving from labeled disjoint sums.

A labeled disjoint sum is called here a *variant*. A variant type looks very much like a record type: it is an unordered set of label-type pairs:

type int_or_bool = [*a*: *int*, *b*: *bool*]

An element of a variant type is a labeled value, where the label is one of the labels in the variant type, and the value has a type matching the type associated to that label. A element of *int_or_bool* is either an integer labeled *int* or a boolean labeled *bool*.

[*a* = 3] : *int_or_bool*

[*b* = *true*] : *int_or_bool*

The basic operations on variants are *is*, which tests whether a variant object has a particular label, and *as*, which extracts the contents of a variant object having a particular label:

[*a* = 3] *is a* ≡ *true*
[*a* = 3] *is b* ≡ *false*
[*a* = 3] *as a* ≡ 3
[*a* = 3] *as b* fails

A variant type σ is a subtype of a variant type τ (written $\sigma \leq \tau$) if τ has all the labels of σ and correspondingly matching types. Hence *int_or_bool* is a subtype of [*a* = *int*, *b* = *bool*,

c = *string*].

When the type associated to a label is *unit* (the trivial type, whose only defined element is *nil*), we can omit the type altogether; a variant type where all fields have *unit* type is also called an enumeration type. The following examples deal with enumeration types.

type precious_metal = [*gold*, *silver*] (i.e. [*gold*: *unit*, *silver*: *unit*])

type metal = [*gold*, *silver*, *steel*]

An object of an enumeration type, e.g. [*gold* = *nil*], can similarly be abbreviated by omitting the "=*nil*" part, e.g. [*gold*].

A function returning a precious metal is also a function returning a metal, hence:

$t \rightarrow precious\_metal \leq t \rightarrow metal$ because $precious\_metal \leq metal$

A function working on metals will also work on precious metals, hence:

$metal \rightarrow t \leq precious\_metal \rightarrow t$ because $precious\_metal \leq metal$

It is evident that [Rule6] holds unchanged for variant types. This justifies the use of the symbol ≤ for both record and variant subtyping. Semantically the subtype relation on variants is mapped to set inclusion, just as in the case of records: *metal* is a set with three defined elements [*gold*], [*silver*] and [*steel*], and *precious_metal* is a set with two defined elements [*gold*] and [*silver*].

There are two ways of deriving variant types from previously defined variant types. We could have defined *metal* and *precious_metal* as:

type precious_metal = [*gold*, *silver*]

type metal = *precious_metal or* [*steel*]

or as:

type metal = [*gold*, *silver*, *steel*]

type precious_metal = *metal dropping steel*

The *or* operator makes a union of the cases of two variant types, and the *dropping* operator removes a case from a variant type. The precise definition of these operators is contained in a later section.

5. Multiple Inheritance

In the framework described so far, we can recognize some of the features of what is called *multiple inheritance* between objects, e.g. a car has (inherits) all the attributes of *vehicle* and of *machine*. Some aspects are however unusual; for example the inheritance relation only depends on the structure of types and need not be declared explicitly.

Moreover, we are not aware of any other language where typechecking coexists with multiple inheritance. Typechecking provides compile-time protection against obvious bugs (like applying the *speed* function to a machine which is not a vehicle), and other less obvious mistakes. Complex type hierarchies can be built where "everything is also something else", and it can be difficult to remember which objects support which messages.

The subtype relation only holds on types, and there is no similar relation on objects. Thus we cannot model directly the *subobject* relation used by, for example, Omega [Attardi 81], where we could define the class of gasoline cars as the cars with fuel equal to "*gasoline*".

However, in simple cases we can achieve the same effect by turning certain sets of values into variant types. For example, instead of having the fuel field of a machine to be a string, we could redefine:

type fueltype = [*coal*, *gasoline*, *electricity*]

```
type machine     = (age: int, fuel: fueltype)
type car         = (age: int, speed: int, fuel: fueltype)
```

Now we can have:

```
type gasoline_car     = (age: int, speed: int, fuel: [gasoline])
type combustion_car   = (age: int, speed: int, fuel: [gasoline, coal])
```

and we have *gasoline_car* ≤ *combustion_car* ≤ *car*. Hence a function over combustion cars, for example, will accept a gasoline car as a parameter, but will give a compile-time type error on electrical cars.

It is often the case that a function which is a field of a record has to refer to other components of the same record. In Smalltalk this is done by referring to the whole record (i.e. object) as *self*, and then selecting the desired components out of that. In Simula there is a similar concept called *this*.

This behavior can be obtained as a special case of the *rec* operator which we are about to introduce. *rec* is used to define recursive functions and data. For example, recursive factorial function can be written as:

```
rec fact: int → int. λn: int. if n=0 then 1 else n*fact(n−1)
```

(This is an expression, not a declaration.)

In order to prevent looping in case of call-by-value evaluations, the body of *rec* is restricted to be a constant, a record, a variant or a function (or, in general, any data constructor present in the language) [Schwarz 80].

Examples of circular data definitions are extremely common in object-oriented programming. In the following example, a functional component of a record refers to "its" other components. The functional component *d*, below, is supposed to compute the distance of "this" *active_point* from any other *point* (or any other *active_point*, etc.).

```
type point = (x: real, y: real)
type active_point = point and (d: point → real)
val make_active_point(px: real, py: real): active_point =
    rec self: active_point.
        (x = px, y = py,
         d = λp: point. sqrt(p.x*self.x + p.y*self.y))
```

Objects often have **private** variables, which are useful to maintain and update the local state of an object while preventing arbitrary external interference. Here is a counter object which starts from some fixed number and can only be incremented one step at a time. *cell n* is an updatable cell whose initial content is *n*; a cell can be updated by := and its contents can be extracted by *get*.

```
type counter = (increment: int → unit, fetch: unit → int)
val make_counter(n: int) =
    let count = cell n
    in  (increment = λn: int. count := (get count)+1,
         fetch = λnil: unit. get count)
```

Private variable are obtained in full generality by the above well known static scoping technique.

6. Expressions

We now begin the formal treatment of multiple inheritance. First, we define a simple applicative language supporting inheritance (side effects could be treated without introducing any new concept, but they make the formal treatment more complicated). Then a denotational semantics is presented, in a domain of values V. Certain subsets of V are regarded as types, and inheritance corresponds directly to set inclusion among types. A type inference system and a typechecking algorithm are then presented. The soundness of the algorithm is proved by showing that the algorithm is consistent with the inference system, and that the inference system is in turn consistent with the semantics.

Our language is typed lambda calculus with records and variants. The following notation is often used for records (and similarly for record and variant types):

$$(a_1{=}e_1, \ldots, a_n{=}e_n) \equiv (a_i{=}e_i) \quad i \in 1..n$$

$$(a_1{=}e_1, \ldots, a_n{=}e_n, a'_1{=}e'_1, \ldots, a'_m{=}e'_m) \equiv (a_i{=}e_i, a'_j{=}e'_j) \quad i \in 1..n,\ j \in 1..m$$

Here is the syntax of expressions and type expressions:

e ::=	expressions	
x |	identifiers	
b |	constants	
if e then e else e |	conditionals	
$(a_i = e_i)$ | $e.a$ |	records	$(i \in 1..n, n \geq 0)$
$[a = e]$ | *e is a* | *e as a* |	variants	
$\lambda x{:}\,\tau.\ e$ | $e\ e$ |	functions	
rec $x{:}\,\tau.\ e$ |	recursive data	
$e{:}\,\tau$ |	type specs	
(e)		

τ ::=	type expressions	
ι |	type constants	
$(a_i{:}\,\tau_i)$ |	record types	$(i \in 1..n, n \geq 0)$
$[a_i{:}\,\tau_i]$ |	variant types	$(i \in 1..n, n \geq 0)$
$\tau \rightarrow \tau$ |	function types	
(τ)		

where $i \neq j \implies a_i \neq a_j$

take $\iota_0 = unit, \quad \iota_1 = bool, \quad \iota_2 = int, \quad$ etc.

Syntactic restriction: the body e of *rec* $x{:}\,\tau.\ e$ can only be a constant, a record, a variant, a lambda expression, or another *rec* obeying this restriction.

Labels a, and identifiers x have the same syntax, but are distinguishable by the syntactic context. Among the type constants we have *unit* (the domain with one defined element) *bool* and *int*. Among the constants we have *nil* (of type *unit*), booleans (*true*, *false*) and numbers (0, 1, ...).

Global definitions of values and types are introduced by the syntax:

d ::=

val $x = e$ \|

type $x = \tau$

where the type definitions are meant as simple abbreviations.

Standard abbreviations are:

let $x{:}\,\tau = e$ *in* e'	for	$(\lambda\, x{:}\,\tau.\ e')\ e$

$f(x\colon \tau)\colon \tau' \;=\; e$ for $f \;=\; \lambda\, x\colon \tau.\ (e\colon \tau')$

$rec\ f(x\colon \tau)\colon \tau' \;=\; e$ for $f \;=\; rec\ f\colon \tau \rightarrowtail \tau'.\ \lambda\, x\colon \tau.\ e$

(the last two abbreviations can only appear after a *let* or a *val*).

Record and variant type expressions are unordered, so for any permutation π_n of $1..n$, we identify:

$(a_i\colon \tau_i) \;\equiv\; (a_{\pi_n(i)}\colon \tau_{\pi_n(i)}) \qquad i \in 1..n$

$[a_i\colon \tau_i] \;\equiv\; [a_{\pi_n(i)}\colon \tau_{\pi_n(i)}] \qquad i \in 1..n$

7. The Semantic Domain

The semantics of expressions is given in the recursively defined domain V of *values*. The domain operators used below are coalesced sum (+), cartesian product ($\times$), continuous function space ($->$) and finite functions ($->_{fin}$, explained later).

$$\begin{aligned} V &= B_0 + B_1 + \cdots + R + U + F + W \\ R &= L \to_{fin} V \\ U &= L \times V \\ F &= V \to V \\ W &= \{\bot, w\} \end{aligned}$$

where L is a countable flat domain of character strings, called *labels*, and B_i are flat domains of basic values. We take:

$$\begin{aligned} B_0 &\equiv O \equiv \{\bot, nil\} \\ B_1 &\equiv T \equiv \{\bot, true, false\} \\ B_2 &\equiv N \equiv \{\bot, 0, 1, \cdots\} \end{aligned}$$

W is a domain which contains a single defined element w, the *wrong* value. The value w is used to model run-time **type errors** (e.g. trying to apply an integer as it were a function) which we want a compiler to trap before execution. It is not used to model run-time **exceptions** (like trying to extract the head of an empty list); in our context these can only be generated by the *as* operator. Run-time exceptions should be modeled by an extra summand of V, but for simplicity we shall instead use the undefined element $\bot$. The name *wrong* is used to denote w as a member of V (instead of simply a member of W).

$R = L \to_{fin} V$ is the domain of *records*, which are associations of values to labels. We are only interested in finite associations, so we define $L \to_{fin} V = \{r \in L \to V \mid \{a \mid r(a) \neq wrong\}$ is finite$\}$.

$U = L \times V$ is the domain of *variants* which are pairs $<l, v>$ with a label l and a value v.

$F = V \to V$ are the continuous functions from V to V, used to give semantics to lambda expressions.

8. Semantics of Expressions

The semantic function is $\mathscr{E} \in Exp \to Env \to V$, where *Exp* are syntactic expressions according to our grammar, and $Env = Id \to V$ are environments for identifiers. The semantics of basic values is given by $\mathscr{B} \in Exp \to V$, whose obvious definition is omitted; b_{ij} is the j-th element of the basic domain B_i.

$\mathscr{E}[\![x]\!]\nu \;=\; \nu[\![x]\!]$

$\mathscr{E}[\![b_{ij}]\!]\nu \;=\; \mathscr{B}[\![b_{ij}]\!]$

$\mathscr{E}[\![$*if* e *then* e' *else* $e''$$]\!]\nu \;=$

$\quad$ *if* $\mathscr{E}[\![e]\!]\nu \in T$ *then* (*if* $(\mathscr{E}[\![e]\!]\nu \mid T)$ *then* $\mathscr{E}[\![e']\!]\nu$ *else* $\mathscr{E}[\![e'']\!]\nu$) *else wrong*

$$\mathcal{E}[\![(a_1 = e_1, \ldots, a_n = e_n)]\!]\nu =$$
$$\quad \textit{if } \mathcal{E}[\![e_1]\!]\nu \in \boldsymbol{W} \textit{ or } \cdots \textit{ or } \mathcal{E}[\![e_n]\!]\nu \in \boldsymbol{W} \textit{ then wrong}$$
$$\quad \textit{else } (\lambda l.\ \textit{if } l{=}a_1 \textit{ then } \mathcal{E}[\![e_1]\!]\nu \textit{ else } \cdots \textit{ if } l{=}a_n \textit{ then } \mathcal{E}[\![e_n]\!]\nu \textit{ else wrong}) \textit{ in } \boldsymbol{V}$$

$$\mathcal{E}[\![e.a]\!]\nu = \textit{if } \mathcal{E}[\![e]\!]\nu \in \boldsymbol{R} \textit{ then } (\mathcal{E}[\![e]\!]\nu \mid \boldsymbol{R})(a) \textit{ else wrong}$$

$$\mathcal{E}[\![[a{=}e]]\!]\nu = \textit{if } \mathcal{E}[\![e]\!]\nu \in \boldsymbol{W} \textit{ then wrong else } \langle a, \mathcal{E}[\![e]\!]\nu\rangle \textit{ in } \boldsymbol{V}$$

$$\mathcal{E}[\![e \textit{ is } a]\!]\nu = \textit{if } \mathcal{E}[\![e]\!]\nu \in \boldsymbol{U} \textit{ then } \textit{fst}(\mathcal{E}[\![e]\!]\nu \mid \boldsymbol{U}) = a \textit{ else wrong}$$

$$\mathcal{E}[\![e \textit{ as } a]\!]\nu =$$
$$\quad \textit{if } \mathcal{E}[\![e]\!]\nu \in \boldsymbol{U} \textit{ then } (\textit{let } \langle b,\nu\rangle \textit{ be } (\mathcal{E}[\![e]\!]\nu \mid \boldsymbol{U}) \textit{ in if } b{=}a \textit{ then } \nu \textit{ else } \bot) \textit{ else wrong}$$

$$\mathcal{E}[\![\lambda x{:}\,\tau.\ e]\!]\nu = (\lambda \nu.\ \mathcal{E}[\![e]\!]\nu\{\nu/[\![x]\!]\}) \textit{ in } \boldsymbol{V}$$

$$\mathcal{E}[\![e\ e']\!]\nu =$$
$$\quad \textit{if } \mathcal{E}[\![e]\!]\nu \in \boldsymbol{F} \textit{ then } (\textit{if } \mathcal{E}[\![e']\!]\nu \in \boldsymbol{W} \textit{ then wrong else } (\mathcal{E}[\![e]\!]\nu \mid \boldsymbol{F})(\mathcal{E}[\![e']\!]\nu)) \textit{ else wrong}$$

$$\mathcal{E}[\![\textit{rec } x{:}\,\tau.\ e]\!]\nu = Y((\lambda \nu.\ \mathcal{E}[\![e]\!]\nu\{\nu/[\![x]\!]\}) \textit{ in } \boldsymbol{V})$$

$$\mathcal{E}[\![e{:}\,\tau]\!]\nu = \mathcal{E}[\![e]\!]\nu$$

Comments on the equations:

- d *in* $\boldsymbol{V}$ (where $d \in \boldsymbol{D}$ and $\boldsymbol{D}$ is a summand of $\boldsymbol{V}$) is the injection of d in the appropriate summand of $\boldsymbol{V}$. Hence d *in* $\boldsymbol{V} \in \boldsymbol{V}$ and $\bot$ *in* $\boldsymbol{V} = \bot$. This is not to be confused with the *let...be...in...* notation for local variables.
- $\nu \in \boldsymbol{D}$ (where $\nu \in \boldsymbol{V}$ and $\boldsymbol{D}$ is a summand of $\boldsymbol{V}$) is a function yielding: $\bot$ if $\nu = \bot$; *true* if $\nu = d$ *in* $\boldsymbol{V}$ for some $d \in \boldsymbol{D}$; *false* otherwise.
- $\nu \mid \boldsymbol{D}$ (where $\boldsymbol{D}$ is a summand of $\boldsymbol{V}$) is a function yielding: d if $\nu = d$ *in* $\boldsymbol{V}$ for some $d \in D$; $\bot$ otherwise.
- *fst* extracts the first element of a pair, *snd* extracts the second one.
- $\mathcal{E}$ defines a call by value semantics.

Intuitively, a well-typed program will never return the *wrong* value at run-time. For example, consider the second occurrence of *wrong* in the semantics of records. The typechecker will make sure that any record selection will operate on records having the appropriate field, hence that instance of *wrong* will never be returned. A similar reasoning applies to all the instances of *wrong* in the semantics: *wrong* is a run-time type error which can be detected at compile-time. Run-time exceptions which cannot be detected are represented as $\bot$; the only instance of this in the above semantics is in the equation for *e as a*.

Formally, we proceed by defining $\mathcal{E}$ (so that it satisfies the above intuitions about run-time errors), then we define "*e* is semantically well-typed" to mean "$\mathcal{E}[\![e]\!]\nu \neq$ *wrong*", and later we give an algorithm which statically checks well-typing.

9. Semantics of Type Expressions

The semantics of types is given in the *weak ideal model* [MacQueen 84] $\mathcal{I}(\boldsymbol{V})$ (the set of non-empty weak ideals which are subset of $\boldsymbol{V}$ and do not contain *wrong*). $\mathcal{I}(\boldsymbol{V})$ is a lattice of domains, where the ordering is set inclusion. $\mathcal{I}(\boldsymbol{V})$ is closed under union and intersection, as well as the usual domain operations.

$$\mathcal{D}[\![\iota_i]\!] = \boldsymbol{B}_i \textit{ in } \boldsymbol{V}$$
$$\mathcal{D}[\![(a_i{:}\,\tau_i)]\!] = \bigcap_i \{r \in \boldsymbol{R} \mid r(a_i) \in \mathcal{D}[\![\tau_i]\!]\} \textit{ in } \boldsymbol{V} \qquad \text{(where we take } \mathcal{D}[\![()]\!] = \boldsymbol{R} \textit{ in } \boldsymbol{V})$$
$$\mathcal{D}[\![[a_i{:}\,\tau_i]]\!] = \bigcup_i \{\langle a_i, \nu\rangle \in \boldsymbol{U} \mid \nu \in \mathcal{D}[\![\tau_i]\!]\} \textit{ in } \boldsymbol{V} \qquad \text{(where we take } \mathcal{D}[\![[]]\!] = \{\bot\})$$
$$\mathcal{D}[\![\sigma \to \tau]\!] = \{f \in \boldsymbol{F} \mid \nu \in \mathcal{D}[\![\sigma]\!] \Longrightarrow f(\nu) \in \mathcal{D}[\![\tau]\!]\} \textit{ in } \boldsymbol{V}$$

where $\mathcal{D}$ *in* $\boldsymbol{V} = \{d \textit{ in } \boldsymbol{V} \mid d \in \boldsymbol{D}\}$

The *wrong* value is deliberately left out of the type domains so that if a value has a type, then that value is not a run-time type error. Another way of saying this is that *wrong* has no type.

10. Type Inclusion

A subtyping relation can be defined syntactically on the structure of type expressions. This definition formalizes our initial discussion of subtyping for records, variants and functions.

$\iota_i \leq \iota_i$

$(a_i\colon \sigma_i,\ a_j\colon \sigma_j) \leq (a_i\colon \sigma'_i) \quad <=> \quad \sigma_i \leq \sigma'_i \qquad (i \in 1..n,\ n \geq 0;\ j \in 1..m,\ m \geq 0)$

$[a_i\colon \sigma_i] \leq [a_i\colon \sigma'_i,\ a_j\colon \sigma'_j] \quad <=> \quad \sigma_i \leq \sigma'_i \qquad (i \in 1..n,\ n \geq 0;\ j \in 1..m,\ m \geq 0)$

$\sigma \to \tau \ \leq\ \sigma' \to \tau' \quad <=> \quad \sigma' \leq \sigma \ \text{ and } \ \tau \leq \tau'$

no other type expressions are in the $\leq$ relation

As we said, the ordering of domains in the $\mathcal{P}(V)$ model is set inclusion. This allows us to give a very direct semantics to subtyping, as simple set inclusion of domains.

THEOREM (Semantic Subtyping)

$$\tau \leq \tau' \implies \mathcal{D}[\![\tau]\!] \subseteq \mathcal{D}[\![\tau']\!]$$

The proof is by induction on the structure of τ and τ'.

11. References

[Ait-Kaci 83] H.Ait-Kaci: "Outline of a calculus of type subsumptions", Technical report MS-CIS-83-34, Dept of Computer and Information Science, The Moore School of Electrical Engineering, University of Pennsylvania, August 1983.

[Albano 83] A.Albano, L.Cardelli, R.Orsini: "Galileo: a strongly typed, interactive conceptual language", Technical Memorandum TM-83-11271-2, Bell Labs, 1983.

[Attardi 81] G.Attardi, M.Simi: "Semantics of inheritance and attributions in the description system Omega", M.I.T. A.I. Memo 642, August 81.

[Borgida 82] A.T.Borgida, J.Mylopulos, H.K.T.Wong: "Methodological and computer aids for interactive information systems design", Automated Tools for Information System Design, H.J.Schneider and A.Wasserman (eds), North-Holland, Amsterdam, 1982.

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

[Deutsch 84] P.Deutsch: "An efficient implementation of Smalltalk-80", Proc. Popl 84.

[Goldberg 83] A.Goldberg, D.Robson: "Smalltalk-80. The language and its implementation", Addison-Wesley, 1983.

[Krasner 83] G.Krasner(Ed.): "Smalltalk-80. Bits of history, words of advice", Addison-Wesley, 1983.

[MacQueen 84] D.B.MacQueen, R.Seti, G.D.Plotkin: "An ideal model for recursive polymorphic types", Proc. Popl 84.

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

[Mitchell 84] J.C.Mitchell: "Coercion and type inference", Proc. Popl 84.

[Steels 83] L.Steels: "Orbit: an applicative view of object-oriented programming", in: Integrated Interactive Computing Systems, pp. 193-205, P.Degano and E.Sandewall editors, North-Holland 1983.

[Weinreb 81] D.Weinreb, D.Moon: "Lisp machine manual", Fourth Edition, Chapter 20: "Objects, Message Passing, and Flavors", Symbolics Inc., 1981.