Luca Cardelli, A Semantics of Multiple Inheritance, in Semantics of Data Types, Lecture Notes in Computer Science 173, 1984. Also to appear in Information and Control.

A Semantics of Multiple Inheritance

Luca Cardelli

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

1. Introduction

There are two major ways of structuring data in programming languages. The first and common one, used for example in Pascal, can be said to derive from standard branches of mathematics. Data is organized as cartesian products (i.e. record types), disjoint sums (i.e. unions or variant types) and function spaces (i.e. functions and procedures).

The second method can be said to derive from biology and taxonomy. Data is organized in a hierarchy of classes and subclasses, and data at any level of the hierarchy *inherits* all the attributes of data higher up in the hierarchy. The top level of this hierarchy is usually called the class of all "objects"; every datum *is an* object and every datum *inherits* the basic properties of objects, like the ability to tell whether two objects are the same or not. Functions and procedures are also considered as local actions of objects, as opposed to global operations.

These different ways of structuring data have generated distinct classes of programming languages, and induced different programming styles. Programming with taxonomically organized data is often called *object-oriented programming*, and has been advocated as an effective way of structuring programming environments, data bases, and large systems in general.

The notions of inheritance and object-oriented programming first appeared in Simula 67 [Dahl 66]. In Simula, objects are grouped into classes and classes can be organized into a subclass hierarchy. Objects are similar to records with functions as 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 implemented as "message-passing" between processes.

Smalltalk [Goldberg 83] adopts and exploits the idea of inheritance, with some changes. While stressing the message-passing paradigm, a Smalltalk object is not usually a separate process. 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. A class can be sometime 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 [Borning 82, Bobrow 83, Hullot 83, Steels 83, Weinreb 81], or as part of knowledge representation languages [Attardi 81]. Exceptions are Galileo [Albano 83] and

OBJ [Futatsugi 85] where multiple inheritance is typechecked.

The definition of what makes a language object-oriented is still controversial. However, 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 which may or may not be present in languages which are commonly considered object-oriented. Hence, a theory of object-oriented programming should first of all focus on the meaning of inheritance.

The aim of this paper is to present a clean semantics of multiple inheritance and to show that, in the context of strongly-typed, statically-scoped languages, a sound typechecking algorithm exists. Multiple inheritance is also interpreted in a broad sense: instead of being limited to objects, it is extended in a natural way to union types and to higher-order functional types. This constitutes a semantic basis for the unification of functional and object-oriented programming.

A clean semantics has the advantage of making clear which issues are fundamental and which are implementation accidents or optimizations. The implementation of multiple inheritance suggested by the semantics is very naive, but does not preclude more sophisticated implementation techniques. It should however be emphasized that advanced implementation techniques are absolutely essential to obtain usable systems based on inheritance [Deutsch 84].

The first part of this paper is informal, and presents the basic notations and intuitions by means of examples. The second part is formal: it introduces a language, a semantics, a type inference system and a typechecking algorithm. The algorithm is proved sound with respect to the inference system, and the inference system is proved sound with respect to the semantics [Milner 78].

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:

$$<a = 3,\ b = true,\ c = \text{"}abc\text{"}>$$

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:

$$<a = 3,\ b = true,\ c = \text{"}abc\text{"}> .\ a \ \equiv\ 3$$

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 definitions are prefixed by the keyword *type*>:

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 $\le$) 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 $\le$ relation. Moreover, all the basic types (like *int* and *bool*) are subtypes of themselves:

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

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

$$val\ mycar = <age = 4,\ speed = 140,\ fuel = \text{"}gasoline\text{"}>$$

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 $\tau \le \tau'$ 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 $a : \sigma$; then it is certainly meaningful to compute $f(a)$ as $f: \sigma \to \tau$.

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

$serial\_number: int \to car$

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 $t \to vehicle$:

$t \to car \leq t \to vehicle$ because $car \leq vehicle$

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 $car \to int$:

$vehicle \to t \leq car \to t$ because $car \leq vehicle$

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

$car \rightarrow int$, [Rule6] hence $(mycar\_attribute: (car \rightarrow int) \rightarrow int)(age: object \rightarrow int)$ 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 to 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:

$$\begin{array}{lll} type\ object & = & \langle age: int \rangle \\ type\ vehicle & = & \langle age: int,\ speed: int \rangle \\ type\ machine & = & \langle age: int,\ fuel: string \rangle \\ type\ car & = & \langle age: int,\ speed: int,\ fuel: string \rangle \end{array}$$

$$\begin{array}{lll} type\ object & = & \langle age: int \rangle \\ type\ vehicle & = & object\ and\ \langle speed: int \rangle \\ type\ machine & = & object\ and\ \langle fuel: string \rangle \\ type\ car & = & vehicle\ and\ machine \end{array}$$

$$\begin{array}{lll} type\ object & = & \langle age: int \rangle \\ type\ car & = & object\ and\ \langle speed: int,\ fuel: string \rangle \\ type\ vehicle & = & car\ ignoring\ fuel \\ type\ machine & = & car\ ignoring\ speed \end{array}$$

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 meet 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, enclosed in brackets:

$$type\ int\_or\_bool\ =\ [a: int,\ b: bool]$$

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 with that label. A element of *int_or_bool* is either an integer labeled a or a boolean labeled b.

$$[a = 3] : int\_or\_bool$$

$$[b = true] : int\_or\_bool$$

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:

$$[a = 3] \; is \; a \quad \equiv \; true$$

$$[a = 3] \; is \; b \quad \equiv \; false$$

$$[a = 3] \; as \; a \quad \equiv \; 3$$

$$[a = 3] \; as \; b \quad \text{fails}$$

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 ()), 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.

$$type\ precious\_metal \quad = \; [gold,\ silver] \qquad \text{(i.e. } [gold: unit,\ silver: unit]\text{)}$$

$$type\ metal \quad = \; [gold,\ silver,\ steel]$$

A value of an enumeration type, e.g. $[gold = ()]$, can similarly be abbreviated by omitting the "=()" part, e.g. $[gold]$.

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

$$t \rightarrow precious\_metal \; \leq \; t \rightarrow metal \qquad \text{because} \qquad precious\_metal \leq metal$$

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

$$metal \rightarrow t \; \leq \; precious\_metal \rightarrow t \qquad \text{because} \qquad precious\_metal \leq metal$$

It is evident that [Rule6] holds unchanged for variant types. This justifies the use of the symbol $\leq$ 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. Inheritance idioms

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 objects and need not be declared explicitly. This section shows how to simulate common inheritance techniques. However we are not trying to *explain* existing inheritance schemes (e.g. Smalltalk) in detail, but rather trying to present a new perspective on the problems.

Some differences between this and other inheritance schemes result in net gains. For example, we are not aware of languages where typechecking coexists with multiple inheritance and higher order functions, with the exception of Galileo [Albano 84] and Amber [Cardelli 84] which were developed in conjunction with this work. 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 be a string, we could redefine:

$$\mathit{type\ fueltype} = [\mathit{coal},\ \mathit{gasoline},\ \mathit{electricity}]$$

$$\mathit{type\ machine} = \langle \mathit{age}: \mathit{int},\ \mathit{fuel}: \mathit{fueltype} \rangle$$

$$\mathit{type\ car} = \langle \mathit{age}: \mathit{int},\ \mathit{speed}: \mathit{int},\ \mathit{fuel}: \mathit{fueltype} \rangle$$

Now we can have:

$$\mathit{type\ gasoline\_car} = \langle \mathit{age}: \mathit{int},\ \mathit{speed}: \mathit{int},\ \mathit{fuel}: [\mathit{gasoline}] \rangle$$

$$\mathit{type\ combustion\_car} = \langle \mathit{age}: \mathit{int},\ \mathit{speed}: \mathit{int},\ \mathit{fuel}: [\mathit{gasoline},\ \mathit{coal}] \rangle$$

and we have $\mathit{gasoline\_car} \leq \mathit{combustion\_car} \leq \mathit{car}$. Hence a function over combustion cars, for example, will accept a gasoline car as a parameter, but will give a compile-time type error when applied to electrical cars.

It is often the case that a function contained in a record field 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 self-referential capability 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, the recursive factorial function can be written as:

$$\mathit{rec\ fact}: \mathit{int} \rightarrow \mathit{int}.\ \lambda n: \mathit{int}.\ \mathit{if}\ n=0\ \mathit{then}\ 1\ \mathit{else}\ n * \mathit{fact}(n-1)$$

(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) [Morris 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.).

$$\mathit{type\ point} = \langle x: \mathit{real},\ y: \mathit{real} \rangle$$

$$\mathit{type}\ \mathit{active\_point} = \mathit{point}\ \mathit{and}\ \langle d : \mathit{point} \rightarrow \mathit{real} \rangle$$

$$\begin{aligned}
&\mathit{val}\ \mathit{make_active_point}(px : \mathit{real},\ py : \mathit{real}) : \mathit{active_point} = \\
&\quad \mathit{rec}\ \mathit{self} : \mathit{active_point}. \\
&\qquad \langle x = px,\ y = py, \\
&\qquad\ d = \lambda p : \mathit{point}.\ \mathit{sqrt}((p.x - \mathit{self}.x)^{**}2 + (p.y - \mathit{self}.y)^{**}2) \rangle
\end{aligned}$$

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 contents is *n*; a cell can be updated by := and its contents can be extracted by *get*.

$$\mathit{type}\ \mathit{counter} = \langle \mathit{increment} : \mathit{unit} \rightarrow \mathit{unit},\ \mathit{fetch} : \mathit{unit} \rightarrow \mathit{int} \rangle$$

$$\begin{aligned}
&\mathit{val}\ \mathit{make_counter}(n : \mathit{int}) = \\
&\quad \mathit{let}\ \mathit{count} = \mathit{cell}\ n \\
&\quad \mathit{in} \quad \langle \mathit{increment} = \lambda().\ \mathit{count} := (\mathit{get}\ \mathit{count}) + 1, \\
&\qquad\quad \mathit{fetch} = \lambda().\ \mathit{get}\ \mathit{count} \rangle
\end{aligned}$$

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

In the presence of side effects, it can be useful to be able to cascade operations on objects. For example we might want to define a different kind of counter, which could be used in the following way:

$$\mathit{make\_counter}(0).\mathit{increment}().\mathit{increment}().\mathit{fetch}() \quad \equiv \quad 2$$

In this case, a local record operation must be able to return "its" record. This requires both recursive objects and recursive types:

$$\mathit{type}\ \mathit{counter} = \mathit{rec}\ \mathit{counter}.\ \langle \mathit{increment} : \mathit{unit} \rightarrow \mathit{counter},\ \mathit{fetch} : \mathit{unit} \rightarrow \mathit{int} \rangle$$

$$\begin{aligned}
&\mathit{val}\ \mathit{make_counter}(n : \mathit{int}) = \\
&\quad \mathit{let}\ \mathit{count} = \mathit{cell}\ n \\
&\quad \mathit{in} \quad \mathit{rec}\ \mathit{self} : \mathit{counter}. \\
&\qquad\qquad \langle \mathit{increment} = \lambda().\ \mathit{count} := (\mathit{get}\ \mathit{count}) + 1;\ \mathit{self}, \\
&\qquad\qquad\ \mathit{fetch} = \lambda().\ \mathit{get}\ \mathit{count} \rangle
\end{aligned}$$

where ";" is sequencing of operations.

In Smalltalk terminology, a subclass automatically inherits the methods of all its superclasses. A subclass can also redefine inherited methods. In any case all the objects created as members of a particular class or subclass will share the same methods. Here is an example where a class called *Class_A* is defined to have methods *f* and *g*; a *make_A* function creates objects of class *Class_A* by forming records with *f* and *g* components.

$$\begin{aligned}
&\mathit{type}\ \mathit{Class_A} = \langle f : X \rightarrow X',\ g : Y \rightarrow Y' \rangle \\
&\mathit{val}\ \mathit{fOfA}(a : X) : X' = \cdots \\
&\mathit{val}\ \mathit{gOfA}(a : Y) : Y' = \cdots \\
&\mathit{val}\ \mathit{make_A}() : \mathit{Class_A} = \langle f = \mathit{fOfA},\ g = \mathit{gOfA} \rangle
\end{aligned}$$

Now we define a subclass of *Class_A*, called *A_Subclass_B*, which has an extra *h* method. The *make_B* function assembles objects of the subclass from the *f* component of the superclass, explicitly inheriting it, a newly defined *g* component, modifying an inherited method, and a new *h* component, local to the subclass.

$$\begin{aligned}
&\mathit{type}\ \mathit{A_Subclass_B} = \mathit{Class_A}\ \mathit{and}\ \langle h : Z \rightarrow Z' \rangle \\
&\mathit{val}\ \mathit{gOfB}(a : Y) : Y' = \cdots \\
&\mathit{val}\ \mathit{hOfB}(a : Z) : Z' = \cdots \\
&\mathit{val}\ \mathit{make_B}() : \mathit{A_Subclass_B} = \langle f = \mathit{fOfA},\ g = \mathit{gOfB},\ h = \mathit{hOfB} \rangle
\end{aligned}$$

Contrarily to Simula and Smalltalk, nothing prevents having totally different methods in different objects of the same class, as long as those methods have the prescribed type.

On the other hand, both Simula and Smalltalk allow objects to access methods of their superclasses. This cannot be simulated in any general and direct way in our framework, partially because of the presence of multiple superclasses.

6. Typechecking anomalies

The style of inheritance typechecking we have presented has a few unexpected aspects. These have to do with the lack of parametric polymorphism and with side effects.

Consider the following identity function on records with an integer a component:

$$\mathit{type}\ A = \langle a: \mathit{int} \rangle$$
$$\mathit{val}\ \mathit{id}(x: A): A = x$$

It is possible to apply id to a subtype B of A, but type information is lost in the process, as the result will have type A, not B. For example, the following expression will not typecheck:

$$(\mathit{id}(\langle a = 3,\ b = \mathit{true} \rangle)).b$$

While this does not have serious consequences in practice, one is forced to adopt a less polymorphic style than one would like: in the previous example it is necessary to write many identity functions for different types.

The following example shows that inheritance polymorphism can sometime achieve the effect of parametric polymorphism, but not quite:

$$\mathit{type}\ \mathit{anyList} = \mathit{rec}\ \mathit{list}.\ [\mathit{nil}: \mathit{unit},\ \mathit{cons}: \langle \mathit{rest}: \mathit{list} \rangle]$$
$$\mathit{type}\ \mathit{intList} = \mathit{rec}\ \mathit{list}.\ [\mathit{nil}: \mathit{unit},\ \mathit{cons}: \langle \mathit{first}: \mathit{int},\ \mathit{rest}: \mathit{list} \rangle]$$
$$\mathit{type}\ \mathit{intPairList} = \mathit{rec}\ \mathit{list}.\ [\mathit{nil}: \mathit{unit},\ \mathit{cons}: \langle \mathit{first}: \mathit{int},\ \mathit{second}: \mathit{int},\ \mathit{rest}: \mathit{list} \rangle]$$

$$\mathit{val}\ \mathit{rest}(l: \mathit{anyList}): \mathit{anyList} = (l\ \mathit{as}\ \mathit{cons}).\mathit{rest}$$
$$\mathit{val}\ \mathit{intFirst}(l: \mathit{intList}): \mathit{int} = (l\ \mathit{as}\ \mathit{cons}).\mathit{first}$$
$$\mathit{val}\ \mathit{intSecond}(l: \mathit{intPairList}): \mathit{int} = (l\ \mathit{as}\ \mathit{cons}).\mathit{second}$$

$$\mathit{val}\ \mathit{rec}\ \mathit{length}(l: \mathit{anyList}): \mathit{int} =$$
$$\quad \mathit{if}\ l\ \mathit{is}\ \mathit{nil}\ \mathit{then}\ 0\ \mathit{else}\ 1 + \mathit{length}(\mathit{rest}\ l)$$

Here *intPairList* is a subtype of *intList*, which is a subtype of *anyList*. The *rest* operator can work on any of these lists, and it can be used to define a polymorphic *length* function. But it is not possible to define a polymorphic *first* operator. The *intFirst* function above works on *intList* and *intPairList*, and *intSecond* works only on *intPairList*.

Inheritance typechecking has to be restricted to preserve soundness in presence of side effects. Note that parametric polymorphism also has to be restricted for side effects, but the problem seems to be rather different in nature. Consider the following example (due to Antonio Albano), where we assume that it is possible to update record fields by a := operator:

$$\mathit{val}\ f(r: \langle a: \langle\rangle \rangle): \mathit{unit} = r.a := \langle\rangle$$
$$\mathit{val}\ r = \langle a = \langle b = 3 \rangle\rangle$$
$$f(r)$$
$$r.a.b$$

The last expression will cause a run-time error, as the a component of r has been changed to $\langle\rangle$ by f. To prevent this, it is sufficient to distinguish syntactically between updatable and non-updatable record fields, and to require type equivalence (instead of type inclusion) while checking inclusion of updatable fields.

7. Expressions

We now begin the formal treatment of multiple inheritance. First, we define a simple applicative language supporting inheritance. 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 $\quad i \neq j \implies a_i \neq a_j$

take $\quad \iota_0 = unit, \quad \iota_1 = bool, \quad \iota_2 = int,$ 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 () (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	$(\lambda\, x : \tau .\, e')\, e$
$f(x : \tau) : \tau' = e$	for	$f = \lambda\, x : \tau .\, (e : \tau')$

$$rec\ f\,(x:\tau):\tau' \ = \ e \qquad \text{for} \qquad f \ = \ rec\ f:\tau\to\tau'.\ \lambda\ x:\tau.\ e$$

(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:

$$<a_i:\tau_i> \ \equiv \ <a_{\pi_n(i)}:\tau_{\pi_n(i)}> \quad i \in 1..n$$

$$[a_i:\tau_i] \ \equiv \ [a_{\pi_n(i)}:\tau_{\pi_n(i)}] \quad i \in 1..n$$

8. 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).

$$\begin{aligned} V &= B_0 + B_1 + \cdots + R + U + F + W \\ R &= L ->_{fin} V \\ U &= L \times V \\ F &= V -> V \\ W &= \{\perp, w\} \end{aligned}$$

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

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

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 if 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 $\perp$. The name *wrong* is used to denote w as a member of V (instead of simply a member of W).

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

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

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

9. Semantics of Expressions

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

$$\mathcal{E}[\![x]\!]\nu = \nu[\![x]\!]$$

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

$$\mathcal{E}[\![if\ e\ then\ e'\ else\ e'']\!]\nu =$$

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

$$\mathcal{E}[\![<a_1 = e_1, \ldots, a_n = e_n>]\!]\nu =$$

$$\quad if\ \mathcal{E}[\![e_1]\!]\nu \in W\ or\ \cdots\ or\ \mathcal{E}[\![e_n]\!]\nu \in W\ then\ wrong$$

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

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

$$\mathscr{E}[\![[a=e]]\!]\nu \;=\; \textit{if } \mathscr{E}[\![e]\!]\nu \in \mathbf{W} \textit{ then wrong else } <a, \mathscr{E}[\![e]\!]\nu> \textit{ in } \mathbf{V}$$

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

$$\mathscr{E}[\![e \textit{ as } a]\!]\nu \;=$$
$$\textit{if } \mathscr{E}[\![e]\!]\nu \in \mathbf{U} \textit{ then } (\textit{let } <b,v> \textit{ be } (\mathscr{E}[\![e]\!]\nu \mid \mathbf{U}) \textit{ in if } b=a \textit{ then } v \textit{ else } \bot) \textit{ else wrong}$$

$$\mathscr{E}[\![\lambda x : \tau.\ e]\!]\nu \;=\; (\lambda v.\ \mathscr{E}[\![e]\!]\nu\{v/[\![x]\!]\}) \textit{ in } \mathbf{V}$$

$$\mathscr{E}[\![e\ e']\!]\nu \;=$$
$$\textit{if } \mathscr{E}[\![e]\!]\nu \in \mathbf{F} \textit{ then } (\textit{if } \mathscr{E}[\![e']\!]\nu \in \mathbf{W} \textit{ then wrong else } (\mathscr{E}[\![e]\!]\nu \mid \mathbf{F})(\mathscr{E}[\![e']\!]\nu)) \textit{ else wrong}$$

$$\mathscr{E}[\![\textit{rec } x : \tau.\ e]\!]\nu \;=\; Y((\lambda v.\ \mathscr{E}[\![e]\!]\nu\{v/[\![x]\!]\}) \textit{ in } \mathbf{V})$$

$$\mathscr{E}[\![e : \tau]\!]\nu \;=\; \mathscr{E}[\![e]\!]\nu$$

Comments on the equations:

- d *in* $\mathbf{V}$ (where $d \in \mathbf{D}$ and $\mathbf{D}$ is a summand of $\mathbf{V}$) is the injection of d in the appropriate summand of $\mathbf{V}$. Hence d *in* $\mathbf{V} \in \mathbf{V}$ and $\bot$ *in* $\mathbf{V} = \bot$. This is not to be confused with the *let...be...in...* notation for local variables.
- $\nu \in \mathbf{D}$ (where $\nu \in \mathbf{V}$ and $\mathbf{D}$ is a summand of $\mathbf{V}$) is a function yielding: $\bot$ if $\nu = \bot$; *true* if $\nu = d$ *in* $\mathbf{V}$ for some $d \in \mathbf{D}$; *false* otherwise.
- $\nu \mid \mathbf{D}$ (where $\mathbf{D}$ is a summand of $\mathbf{V}$) is a function yielding: d if $\nu = d$ *in* $\mathbf{V}$ for some $d \in \mathbf{D}$; $\bot$ otherwise.
- *fst* extracts the first element of a pair, *snd* extracts the second one.
- $\mathscr{E}$ 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 $\mathscr{E}$ (so that it satisfies the above intuitions about run-time errors), then we define "*e* is semantically well-typed" to mean "$\mathscr{E}[\![e]\!]\nu \neq \textit{wrong}$", and later we give an algorithm which statically checks well-typing.

10. Semantics of Type Expressions

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

$$\mathscr{D}[\![\iota_i]\!] \;=\; \mathbf{B}_i \textit{ in } \mathbf{V}$$

$$\mathscr{D}[\![<a_i: \tau_i>]\!] \;=\; \bigcap_i \{r \in \mathbf{R} \mid r(a_i) \in \mathscr{D}[\![\tau_i]\!]\} \textit{ in } \mathbf{V} \qquad \text{(where we take } \mathscr{D}[\![()]\!] = \mathbf{R} \textit{ in } \mathbf{V}\text{)}$$

$$\mathscr{D}[\![[a_i: \tau_i]]\!] \;=\; \bigcup_i \{<a_i, v> \in \mathbf{U} \mid v \in \mathscr{D}[\![\tau_i]\!]\} \textit{ in } \mathbf{V} \qquad \text{(where we take } \mathscr{D}[\![[\,]]\!] = \{\bot\}\text{)}$$

$$\mathscr{D}[\![\sigma \to \tau]\!] \;=\; \{f \in \mathbf{F} \mid v \in \mathscr{D}[\![\sigma]\!] \Longrightarrow f(v) \in \mathscr{D}[\![\tau]\!]\} \textit{ in } \mathbf{V}$$

where $\mathscr{D}$ *in* $\mathbf{V} = \{d \textit{ in } \mathbf{V} \mid d \in \mathbf{D}\}$

THEOREM ($\mathscr{D}$ properties)

$$\forall \tau.\ \bot \in \mathscr{D}[\![\tau]\!]$$

$$\forall \tau, \nu.\ \nu \in \mathscr{D}[\![\tau]\!] \Longrightarrow \nu \neq \textit{wrong}$$

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.

11. 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.

$$\iota_i \le \iota_i$$

$$\langle a_i\colon \sigma_i,\ a_j\colon \sigma_j\rangle \le \langle a_i\colon \sigma'_i\rangle \quad \Longleftrightarrow \quad \sigma_i \le \sigma'_i \qquad (i \in 1..n,\ n \geq 0;\ j \in 1..m,\ m \geq 0)$$

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

$$\sigma \rightarrow \tau \ \le \ \sigma' \rightarrow \tau' \quad \Longleftrightarrow \quad \sigma' \le \sigma \ \text{ and } \ \tau \le \tau'$$

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

PROPOSITION:

$\le$ is a partial order

It is possible to extend type expressions by two constants *anything* and *nothing*, such that *nothing* $\le \tau \le$ *anything* for any τ. Then, $\le$ defines a lattice structure on type expressions, which is a sublattice of $\mathscr{I}(V)$. Although this is mathematically appealing, we have chosen not to do it in view of our intended application. For example, the expression *if x then* 3 *else true*, should produce a type error because of a conflict between *int* and *bool* in the two branches of the conditional. If we have the full lattice of type expression, it is conceivable to return *anything* as the type of the expression above, and carry on typechecking. This is bad for two reasons. First, no use can be made of objects of type *anything* (at least in the present framework). Second, type errors are difficult to localize as their presence is only made manifest by the eventual occurrence of *anything* or *nothing* in the resulting type.

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

THEOREM (Semantic Subtyping)

$$\tau \le \tau' \quad \Longrightarrow \quad \mathfrak{D}[\![\tau]\!] \subseteq \mathfrak{D}[\![\tau']\!]$$

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

12. Type Inference Rules

In this section we formally define the notion of a **syntactically well−typed** expression. An expression is well-typed when a type can be deduced for it, according to a set of type rules forming an **inference system**. If no type can be deduced, then the expression is said to contain type errors.

In general, many types can be deduced for the same expression. Provided that the inference system is consistent, all those types are in some sense compatible. A typechecking algorithm can then choose any of the admissible types as *the* type of an expression, with respect to that algorithm (in some type systems there may be a *best*, or *most general*, or *principal* type). Inference systems can be shown to be consistent with respect to the semantics of the language, as we shall see at the end of this section.

Hence, here is the inference system for our language. It is designed so that (1) it contains exactly one type rule for each syntactic construct; (2) it satisfies the intuitive subtyping property expressed by the syntactic subtyping theorem below; and (3) it satisfies a semantic soundness theorem, relating it to the semantics of the language.

The use of the subtyping predicate $\le$ is critical in many type rules. However it should be noted that subtyping does not affect the fundamental λ-calculus typing rules [ABS] and [COMB]. This indicates that this style of subtyping merges naturally with functional types.

[VAR] $A.x:\tau \vdash x:\tau'$ where $\tau \leq \tau'$

[BAS] $A \vdash b_{ij}: \iota_i$

[COND] $$\frac{A \vdash e: bool \quad A \vdash e': \tau \quad A \vdash e'': \tau}{A \vdash (if\ e\ then\ e'\ else\ e''): \tau}$$

[RECORD] $$\frac{A \vdash e_1: \tau_1 \quad \cdots \quad A \vdash e_n: \tau_n}{A \vdash <a_1 = e_1, \ldots, a_n = e_n): <a_i: \tau_i>}$$ where $i \in I \subseteq 1..n$

[DOT] $$\frac{A \vdash e: < \ldots a: \tau \ldots >}{A \vdash e.a: \tau}$$

[VARIANT] $$\frac{A \vdash e: \tau}{A \vdash [a = e]: [\ \ldots a: \tau \ldots\ ]}$$

[IS] $$\frac{A \vdash e: [\ \ldots a: \sigma \ldots\ ]}{A \vdash (e\ is\ a>: bool}$$

[AS] $$\frac{A \vdash e: [\ \ldots a: \tau \ldots\ ]}{A \vdash (e\ as\ a): \tau}$$

[ABS] $$\frac{A.x: \sigma \vdash e: \tau}{A \vdash (\lambda x: \sigma.\ e): \sigma \rightarrow \tau}$$

[COMB] $$\frac{A \vdash e: \sigma \rightarrow \tau \quad A \vdash e': \sigma}{A \vdash (e\ e'): \tau}$$

[REC] $$\frac{A.x: \sigma \vdash e: \rho}{A \vdash (rec\ x: \sigma.\ e): \tau}$$ where $\rho \leq \sigma$ and $\rho \leq \tau$

[SPEC] $$\frac{A \vdash e: \sigma}{A \vdash (e: \sigma): \tau}$$ where $\sigma \leq \tau$

Some comments on the rules:

- A is a list of assumptions for variables, of the form $x: \tau$. We use the notation $A \equiv A'.x: \tau$ to single out the assumption at the head of A. The handling of nested scopes for variables requires some care: a list of assumptions can be permuted as long as assumptions involving the same variable are not swapped.
- If there are some nontrivial inclusions in the basic types (e.g. *int* $\leq$ *real*) then [BAS] must be changed to $A \vdash b_{ij}: \tau$ where $\iota_i \leq \tau$.
- In [RECORD], the derived record type can have fewer fields than the corresponding record object.
- In [VARIANT], the derived variant type can have any number of fields, as long as it includes a field corresponding to the variant object.
- In [IS], the antecedent could safely be changed to have an arbitrary variant type. It would however seem strange to be able to test the existence of a label but fail to typecheck when trying to extract out of that same label (see [AS]).
- The [IS] rule assumes that the set of basic types does not contain a supertype of *bool*, otherwise a more refined rule is needed. Similarly, [COND] assumes that there are no subtypes of *bool*.

The basic syntactic property of this inference system is expressed in the syntactic subtyping theorem below: if an expression has a type τ, and τ is a subtype of τ', then the expression has also type τ'. The lemma is required to prove the [ABS] case of the theorem. Both the lemma and the theorem are proved by induction on the structure of the derivations.

LEMMA (Syntactic Subtyping):

$$A.x:\sigma \vdash e:\tau \text{ and } \sigma' \leq \sigma \implies A.x:\sigma' \vdash e:\tau$$

THEOREM (Syntactic Subtyping):

$$A \vdash e:\tau \text{ and } \tau \leq \tau' \implies A \vdash e:\tau'$$

The next theorem states the soundness of the type system with respect to the semantics: if it is possible to deduce that e has type τ, then the value denoted by e belongs to the domain denoted by τ. A list of assumptions A agrees with an environment ν if for any x, $\nu[\![x]\!] \in \mathcal{D}[\![\tau]\!] \iff A = A'.x:\tau$.

THEOREM (Semantic Soundness):

$$\text{If } \nu \text{ agrees with } A \text{ and } A \vdash e:\tau \text{ then } \mathcal{E}[\![e]\!]\nu \in \mathcal{D}[\![\tau]\!]$$

The proof is by induction on the structure of the derivation of $A \vdash e:\tau$, using the semantic subtyping and $\mathcal{D}$-properties theorems.

Another way of looking at this theorem is that if e is syntactically well-typed (i.e. for some τ, $A \vdash e:\tau$), then it is also semantically well-typed (i.e. for some τ, $\mathcal{E}[\![e]\!]\nu \in \mathcal{D}[\![\tau]\!]$, which implies that $\mathcal{E}[\![e]\!]\nu \neq \mathit{wrong}$).

13. Join and Meet Types

In the examples at the beginning of the paper we used the *and* and *or* type operators, and we are now going to need them in the definition of the typechecking algorithm. However those operators are not part of the syntax of type expressions, nor are *ignoring* and *dropping*.

This is because the above operators only work on restricted kinds of type expressions. Applied to arbitrary type expressions they either are undefined, or can be eliminated by a normalization process. Hence, if we have a type expression containing the above operators we can process the expression checking that the operators can be indeed used in that context, and in such case we can normalize them away obtaining a normal type expression.

The *and* operator is interpreted as a meet operation on types (written $\downarrow$), and *or* is interpreted as join (written $\uparrow$). Joins and meets are taken in the partial order determined by $\leq$, when they exist.

The definition of the operators also immediately defines the normalization process which eliminates them:

$$\iota_i \uparrow \iota_i = \iota_i$$

$$\langle a_i:\tau_i,\ b_j:\sigma_j\rangle \uparrow \langle a_i:\tau'_i,\ c_k:\rho_k\rangle = \langle a_i:\tau_i \uparrow \tau'_i\rangle \qquad (\forall j,k.\ b_j \neq c_k)$$

$$[a_i:\tau_i,\ b_j:\sigma_j] \uparrow [a_i:\tau'_i,\ c_k:\rho_k] = [a_i:\tau_i \uparrow \tau'_i,\ b_j:\sigma_j,\ c_k:\rho_k] \qquad (\forall j,k.\ b_j \neq c_k)$$

$$(\sigma \to \tau) \uparrow (\sigma' \to \tau') = (\sigma \downarrow \sigma') \to (\tau \uparrow \tau')$$

$$\tau \uparrow \tau' \qquad \text{undefined otherwise}$$

$$\iota_i \downarrow \iota_i = \iota_i$$

$$\langle a_i:\tau_i,\ b_j:\sigma_j\rangle \downarrow \langle a_i:\tau'_i,\ c_k:\rho_k\rangle = \langle a_i:\tau_i \downarrow \tau'_i,\ b_j:\sigma_j,\ c_k:\rho_k\rangle \qquad (\forall j,k.\ b_j \neq c_k)$$

$$[a_i:\tau_i,\ b_j:\sigma_j] \downarrow [a_i:\tau'_i,\ c_k:\rho_k] = [a_i:\tau_i \downarrow \tau'_i] \qquad (\forall j,k.\ b_j \neq c_k)$$

$$(\sigma \to \tau) \downarrow (\sigma' \to \tau') = (\sigma \uparrow \sigma') \to (\tau \downarrow \tau')$$

$$\tau \downarrow \tau' \qquad \text{undefined otherwise}$$

$$<a_i\!: \tau_i> \textit{ ignoring } a \;=\; <a_j\!: \tau_j> \qquad (i \in 1..n,\;\; j \in 1..n - \{k \mid a_k = a\})$$

$$\tau \textit{ ignoring } a \qquad \text{undefined otherwise}$$

$$[a_i\!: \tau_i] \textit{ dropping } a \;=\; [a_j\!: \tau_j] \qquad (i \in 1..n,\;\; j \in 1..n - \{k \mid a_k = a\})$$

$$\tau \textit{ dropping } a \qquad \text{undefined otherwise}$$

PROPOSITION ($\uparrow$ and $\downarrow$ properties):

$\uparrow$ and $\downarrow$ are join and meet for $\leq$, when defined

$$\mathcal{D}[\![\sigma \textit{ and } \tau]\!] \;=\; \mathcal{D}[\![\sigma]\!] \cap \mathcal{D}[\![\tau]\!] \qquad \text{(when defined)}$$

$$\mathcal{D}[\![\tau \textit{ ignoring } a]\!] \;=\; \{r \in R \mid ((r/a) \textit{ in } V) \in \mathcal{D}[\![\tau]\!]\} \textit{ in } V \qquad \text{(when defined)}$$

$$\mathcal{D}[\![\sigma \textit{ or } \tau]\!] \;=\; \mathcal{D}[\![\sigma]\!] \cup \mathcal{D}[\![\tau]\!] \qquad \text{(when defined)}$$

$$\mathcal{D}[\![\tau \textit{ dropping } a]\!] \;=\; \mathcal{D}[\![\tau]\!] - \{<a,\, v> \in U\} \textit{ in } V \qquad \text{(when defined)}$$

where $r/a = (\lambda b.\ \textit{if } b{=}a \textit{ then } \perp \textit{ else } r(b))$.

14. Typechecking

The typechecking function is $\mathcal{T} \in \textit{Exp} \;{-}{>}\; \textit{TypeEnv} \;{-}{>}\; \textit{TypeExp}$, where $\textit{Exp}$ and $\textit{TypeExp}$ are respectively expressions and type expressions according to our grammar, and $\textit{TypeEnv} = \textit{Id} \;{-}{>}\; \textit{TypeExp}$ are type environments for identifiers.

The following description is to be intended as a scheme for a program that returns a type expression denoting the type of a term, or fails in case of type errors. The *fail* word is a global jump-out: when a type error is detected the program stops. Similarly, typechecking fails when the $\uparrow$ and $\downarrow$ operations are undefined. When we assert that $\mathcal{T}[\![e]\!]\mu = \tau$, we imply that the typechecking of e does not fail.

$$\mathcal{T}[\![x]\!]\mu \;=\; \mu[\![x]\!]$$

$$\mathcal{T}[\![b_{ij}]\!]\mu \;=\; \iota_i$$

$$\mathcal{T}[\![\textit{if } e \textit{ then } e' \textit{ else } e'']\!]\mu \;=\; \textit{if } \mathcal{T}[\![e]\!]\mu{=}\textit{bool then } \mathcal{T}[\![e']\!]\mu \uparrow \mathcal{T}[\![e'']\!]\mu \textit{ else fail}$$

$$\mathcal{T}[\![<a_1 = e_1,\, \dots,\, a_n = e_n>]\!]\mu \;=\; <a_1\!: \mathcal{T}[\![e_1]\!]\mu,\, \dots,\, a_n\!: \mathcal{T}[\![e_n]\!]\mu>$$

$$\mathcal{T}[\![e.a]\!]\mu \;=\; \textit{if } \mathcal{T}[\![e]\!]\mu{=}<\dots a : \tau \dots> \textit{ then } \tau \textit{ else fail}$$

$$\mathcal{T}[\![[a = e]]\!]\mu \;=\; [a : \mathcal{T}[\![e]\!]\mu]$$

$$\mathcal{T}[\![e \textit{ is } a]\!]\mu \;=\; \textit{if } \mathcal{T}[\![e]\!]\mu{=}[\dots a : \tau \dots] \textit{ then bool else fail}$$

$$\mathcal{T}[\![e \textit{ as } a]\!]\mu \;=\; \textit{if } \mathcal{T}[\![e]\!]\mu{=}[\dots a : \tau \dots] \textit{ then } \tau \textit{ else fail}$$

$$\mathcal{T}[\![\lambda x : \tau.\ e]\!]\mu \;=\; \tau \rightarrow \mathcal{T}[\![e]\!]\mu\{\tau / [\![x]\!]\}$$

$$\mathcal{T}[\![e\ e']\!]\mu \;=\; \textit{if } \mathcal{T}[\![e]\!]\mu{=}(\tau \rightarrow \tau') \textit{ and } \mathcal{T}[\![e']\!]\mu \leq \tau \textit{ then } \tau' \textit{ else fail}$$

$$\mathcal{T}[\![\textit{rec } x : \sigma.\ e]\!]\mu \;=\; \textit{if } \mathcal{T}[\![e]\!]\mu\{\sigma / [\![x]\!]\}{=}\tau \textit{ and } \tau \leq \sigma \textit{ then } \tau \textit{ else fail}$$

$$\mathcal{T}[\![e : \sigma]\!]\mu \;=\; \textit{if } \mathcal{T}[\![e]\!]\mu{=}\tau \textit{ and } \tau \leq \sigma \textit{ then } \tau \textit{ else fail}$$

This typechecking algorithm is correct with respect to the type inference system: if the algorithm succeeds and returns a type τ for an expression e, then it is possible to prove that e has type τ. A type environment μ agrees with a list of assumptions A if $\mu[\![x]\!] = \tau \iff A = A'.x : \tau$.

THEOREM (Syntactic Soundness):

if A agrees with μ and $\mathcal{T}[\![e]\!]\mu = \tau$ then $A \vdash e : \tau$

The proof is by induction on the structure of e, using the properties of $\uparrow$, $\downarrow$ and $\leq$.

Combining the syntactic soundness, semantic soundness and $\mathfrak{D}$-properties theorems we immediately obtain:

COROLLARY (Typechecking prevents type errors):

$$\text{if } \mathcal{T}[\![e]\!]\mu = \tau \text{ then } \mathcal{E}[\![e]\!]\nu \neq \mathit{wrong} \qquad \text{(where } \mu \text{ agrees with some } A \text{ which agrees with } \nu\text{)}$$

i.e. if e can be successfully typechecked, then e cannot produce run-time type errors.

15. Conclusions

This work originated as an attempt to justify the multiple inheritance constructs present in the Galileo data base language [Albano 83] and to provide a sound typechecking algorithm for that language. The Amber language [Cardelli 84] was then devised to experiment, among other things, with inheritance typechecking. I believe this paper adequately solves the basic problems, although some practical and theoretical issues may require more work.

Parametric polymorphism has not been treated in this paper. The intention was to study multiple inheritance problems in the cleanest possible framework, without interaction of other features. Side-effects and circular types should also be integrated in a full formal treatment.

Some confusion may arise from the fact that languages like Smalltalk are often referred to as polymorphic languages. This is correct, if by polymorphism we mean that an object or a function can have many types. However it now appears that there are two subtly different kinds of polymorphism: inheritance polymorphism, based on type inclusion, and parametric polymorphism, based on type variables and type quantifiers.

These two kinds of polymorphism are not incompatible. We have seen here that inheritance can be explained in the semantic domains normally used for parametric polymorphism. Moreover the technical explanation of polymorphism is the same in both cases: domain intersection. Merging these two kinds of polymorphism does not seem to introduce new semantic problems. The interactions of inheritance and parametric polymorphism in typechecking may however be complex, and this is left here as the major open problem.

There are now several competing (although not totally independent) styles of parametric polymorphism, noticeably [Milner 78], [Reynolds 74, McCracken 84] and [MacQueen 84]. Inheritance is orthogonal to all of these, so it seems better to study it independently, at least initially. However the interactions will have to be investigated in order to obtain the best of all possible worlds.

16. Related work and acknowledgements

I would like to mention here [Reynolds 80, Oles 84] which expose the same basic semantic ideas in a different formal framework, [Ait-Kaci 83] again very similar ideas in a Prolog-related framework, [Mitchell 84] this time different, but related, ideas in the same formal framework, [Futatsugi 85] whose OBJ system implements a first-order multiple inheritance typechecker, and whose subsorts have much to do with subtypes, and [Fairbairn 84] whose quantified-types typechecker may be relevant to unifying inheritance with parametric polymorphism.

Finally, I would like to thank Dave MacQueen for many discussions, and Antonio Albano and Renzo Orsini for motivating me to carry out this work.

17. 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", Bell Labs Technical Memorandum TM 83-11271-2.

[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.

[Bobrow 83] D.G.Bobrow, M.J.Stefik: "The Loops manual", Memo KB-VLSI-81-13, Xerox PARC.

[Cardelli 84] L.Cardelli: "Amber", AT&T Bell Laboratories Technical Memorandum, 11271-840924-10TM, 1984.

[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.

[Futatsugi 85] K.Futatsugi, J.A.Goguen, J.P.Jouannaud, J.Meseguer: "Principles of OBJ2", Proc. POPL '85.

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

[Hullot 83] J-M.Hullot: "Ceyx: a Multiformalism programming environment", IFIP 83, R.E.A.Mason (ed), North Holland, Paris 1983.

[McCracken 84] N.McCracken: "The typechecking of programs with implicit type structure", this conference.

[MacQueen 84] D.B.MacQueen, R.Sethi, 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.

[Oles 84] F.J.Oles: "Type algebras, functor categories, and block structure", to appear in "Algebraic semantics", M.Nivat and J.C.Reynolds ed., Cambridge University Press 1984.

[Fairbairn 84] J.Fairbairn: "A new typechecker for a functional language", Technical report No 53, University of Cambridge, Computer Laboratory.

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

[Reynolds 80] J.C.Reynolds: "Using category theory to design implicit type conversions and generic operators", in "Semantics-directed compiler generation", Lecture Notes in Computer Science 94, pp. 211-258, Springer-Verlag 1980.

[Morris 80] L.Morris, J.Schwarz: "Computing cyclic list structures", Conference Record of the 1980 Lisp Conference, pp.144-153.

[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.