Information and Computation 76, 138-164, 1988

A Semantics of Multiple Inheritance

Luca Cardelli[1]

AT&T Bell Laboratories, Murray Hill, NJ 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, e.g. the ability to tell whether two objects are the same or not. Functions and procedures are considered as local actions of objects, as opposed to global operations acting over objects.

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 realized by function calls, although the association of message

---

[1] Present address: DEC SRC, 130 Lytton Ave, Palo Alto, CA 94301.

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 sometimes be 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 in describing class hierarchies, but it is 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 85] and OBJ [Futatsugi 85] where multiple inheritance is typechecked.

The definition of what makes a language object-oriented is still controversial. An examination of 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 features 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 naïve, but does not preclude more sophisticated implementation techniques. It should 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 achieved by simple field selection (of functional record components) 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 achieved by 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 the objects-as-records paradigm 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*. However, the duality between records and functions remains: in our language objects are records, but the semantics interprets records as functions.

3. Records

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

```
{a = 3, b = true, c = "abc"}
```

This 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  ≡  3
```

An expression can have one or more types; we write

```
e : τ
```

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 ≤) 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 ≤ relation. Moreover, all the basic types (like int and bool) are subtypes of themselves:

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

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

```
value 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:

```
value age(x: object): int = x.age
```

we can meaningfully compute age(mycar) because, 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): τ
[Rule5] if $f: \sigma \to \tau$ and $a: \sigma'$, where $\sigma' \leq \sigma$ then f(a) is meaningful, and f(a): τ

[Rule5] is just a consequence of [Rule3] and [Rule4]. From [Rule3] and $a: \sigma'$ 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 → car
```

We can argue that serial_number returns vehicles, as all cars are vehicles. In general, all car-valued functions 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 → 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 int. 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 → t ≤ car → t because car ≤ 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 → operator, as explained formally in the following sections. (Semantically, we work in a universal value domain $\boldsymbol{V}$ of all computable values. Every function f is a function from $\boldsymbol{V}$ to $\boldsymbol{V}$, written f: $\boldsymbol{V}$ –> $\boldsymbol{V}$, where –> is the conventional continuous function space. By f: σ→τ we indicate a function f: $\boldsymbol{V}$ –> $\boldsymbol{V}$ which whenever given an element of $\sigma \subseteq \boldsymbol{V}$ returns an element of $\tau \subseteq \boldsymbol{V}$; nothing is asserted about the behavior of f outside σ).

Given any function f: σ→τ 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→vehicle can be used in the context age(f(mycar)), where it is used as a function f: car→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.

```
value mycar_attribute(f: car → int): int = f(mycar)
```

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

```
mycar_attribute(age)      ≡ 4
mycar_attribute(speed)    ≡ 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.

We conclude this section with a pragmatic consideration about record notation. Record types can have a large number of fields, hence we need some way of 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 meet operation on type expressions, as explained in a later section.

The ignoring operator simply eliminates a component from a record type. Both and and ignoring are undefined on types other than record 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. An element of int_or_bool is either an integer value labeled a or a boolean value labeled b.

```
value an_int  = [a = 3] : int_or_bool
value a_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:

```
an_int is a  ≡ true
an_int is b  ≡ false
an_int as a ≡ 3
an_int as b  does not have a value
```

A variant type σ is a subtype of a variant type τ (written σ≤τ) 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 the constant unity), 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]
```

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

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

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

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

$metal \to t \leq precious\_metal \to t$ 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. 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 compares our approach with other approaches to inheritance, and shows how to simulate a number of 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 issues.

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 85] and Amber [Cardelli 86] 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:

```
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 obtain 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 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. The rec operator is used to define recursive functions and data. For example, the 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.)

The body of rec is restricted to be a *constructor*; this is a vague term indicating that, in an implementation, computation can be temporarily suspended thereby avoiding some looping situations [Morris 80]. In the language we are considering, a constructor is either a constant, a record, a variant, a function or a rec expression obeying this restriction.

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, computes the distance of *this* active_point from any other point.

```
type point =
    {x: real, y: real}
type active_point =
    point and {d: point → real}
value 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)**2 + (p.y - self.y)**2)}
```

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 (side-effects will not be treated in the formal semantics). Here, λ().e is an abbreviation for λx:unit.e, where x does not occur in e, and let x = a in b introduces a new variable x (initialized to a) local to the scope of b, whose value is returned.

```
type counter =
    {increment: unit → unit, fetch: unit → int}
value make_counter(n: int): counter =
    let count = cell n
    in  {increment = λ(). count := (get count)+1,
         fetch = λ(). get count}
```

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 cascade operations on objects. For example we might want to define a different kind of counter, which could be used in the following way (where f() is an abbreviation for f(unity)):

```
make_counter(0).increment().increment().fetch()  ≡  2
```

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

```
type counter =
    rec counter. {increment: unit → counter, fetch: unit → int}
value make_counter(n: int): counter =
    let count = cell n
    in  rec self: counter.
            {increment = λ(). count := (get count)+1; self,
             fetch = λ(). get count}
```

where ";" is sequencing of operations. (Recursive types will not be treated in the formal semantics; we believe they can be dealt with, but the complications would distract us from the major topic of this paper.)

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.

```
type Class_A                = {f: X → X', g: Y → Y'}
value fOfA(a: X): X'        =  ...
value gOfA(a: Y): Y'        =  ...
value make_A(): Class_A     = {f = fOfA, g = gOfA}
```

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.

```
type A_Subclass_B           = Class_A and {h: Z → Z'}
value gOfB(a: Y): Y'        =  ...
value hOfB(a: Z): Z'        =  ...
value make_B(): A_Subclass_B = {f = fOfA, g = gOfB, h = hOfB}
```

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

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 having an integer component a:

```
type A              = {a: int}
value 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:

```
(id({a = 3, b = true})).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:

```
type anyList        = rec list. [nil: unit, cons: {rest: list}]
type intList        = rec list. [nil: unit, cons: {first: int, rest: list}]
type intPairList    = rec list. [nil: unit, cons: {first: int, second: int, rest: list}]

value rest(l: anyList): anyList         = (l as cons).rest
value intFirst(l: intList): int         = (l as cons).first
value intSecond(l: intPairList): int    = (l as cons).second

value rec length(l: anyList): int =
    if l is nil then 0 else (1 + length(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. A solution to this problem is proposed in [Cardelli 85], where multiple inheritance and parametric polymorphism are merged.

Inheritance typechecking has to be restricted to preserve soundness in presence of side-effects. Parametric polymorphism also has to be restricted in order to deal with 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 (this is a different update mechanism than the one used in the previous section):

```
value f(r: {a: {}}): unit =
    r.a := {}
value r =
    {a = {b = 3}}
f(r)
r.a.b
```

The last expression will cause a run-time error, as the a component of r has been changed to {} 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. Again, this discussion is informal; side-effects will not be dealt with in the rest of the paper.

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 a variant of the typed lambda calculus with type inclusion, recursion and a data domain including 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 |	$(i \in 1..n, n \geq 0)$	records
[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
$\{a_i : \tau_i\}$ |	$(i \in 1..n, n \geq 0)$	record types
$[a_i : \tau_i]$ |	$(i \in 1..n, n \geq 0)$	variant types
$\tau \rightarrow \tau$ |		function types
(τ)		

where $i \neq j \Rightarrow a_i \neq a_j$
take ι_0 = unit, ι_1 = bool, ι_2 = int, etc.

Syntactic restriction: the body e of rec x: τ. e can only be a constant, a record, a variant, a lambda expression, or another rec expression 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 unity (of type unit), booleans (true, false) and numbers (0, 1, ...).

Instead of the two operations is and as on variants, one could use a single case construct. The former are more direct and illustrate the semantic handling of exceptions, while the latter is more elegant (one construct instead of two) and avoids dealing with exceptions.

Standard abbreviations are (the last two can only appear after a let):

let x: τ = e in e'	for	$(\lambda x: \tau . e')\ e$
f(x: τ): τ' = e	for	f: $\tau \to \tau'$ = $\lambda x: \tau . (e: \tau')$
rec f(x: τ): τ' = e	for	f: $\tau \to \tau'$ = rec f: $\tau \to \tau'$. $\lambda x: \tau . e$

Record and variant type expressions are unordered, so for any permutation $\pi(n)$ 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 $\boldsymbol{V}$ of *values*. The domain operators used below are disjoint sum (+), cartesian product ($\times$), and continuous function space (->).

$$\boldsymbol{V} = \boldsymbol{B}_0 + \boldsymbol{B}_1 + ... + \boldsymbol{R} + \boldsymbol{U} + \boldsymbol{F} + \boldsymbol{W}$$
$$\boldsymbol{R} = \boldsymbol{L} \to \boldsymbol{V}$$
$$\boldsymbol{U} = \boldsymbol{L} \times \boldsymbol{V}$$
$$\boldsymbol{F} = \boldsymbol{V} \to \boldsymbol{V}$$
$$\boldsymbol{W} = \{w\}$$

where $\boldsymbol{L}$ is a flat domain of character strings, called *labels*, and $\boldsymbol{B}_i$ are flat domains of basic values. We take:

$$\boldsymbol{B}_0 \equiv \boldsymbol{O} \equiv \{\perp_{\boldsymbol{O}}, \text{unity}\}$$
$$\boldsymbol{B}_1 \equiv \boldsymbol{T} \equiv \{\perp_{\boldsymbol{T}}, \text{true}, \text{false}\}$$
$$\boldsymbol{B}_2 \equiv \boldsymbol{N} \equiv \{\perp_{\boldsymbol{N}}, 0, 1, ...\}$$

b_{ij} is the j-th element of the basic domain $\boldsymbol{B}_i$

$\boldsymbol{W}$ is a domain which contains a single 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. The name wrong is used to denote w as a member of V (instead of simply a member of W). Run-time exceptions should be modeled by an extra summand of V, but for simplicity we shall instead use the undefined element of V, $\perp_V$ (often abbreviated as $\perp$).

$R = L \to V$

is the domain of *records*, which are associations of values to labels.

$U = L \times V$

is the domain of *variants* which are pairs <l,v> with a label l and a value v.

$F = V \to V$

is the domain of the continuous functions from V to V, used to give semantics to lambda expressions.

9. Semantics of expressions

The semantic function is $\mathbb{E} \in Exp \to Env \to V$, 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 $\mathbb{B} \in Exp \to V$, whose obvious definition is omitted.

Using the conventions below, we define:

$\mathbb{E}[\![x]\!]\eta = \eta[\![x]\!]$

$\mathbb{E}[\![b_{ij}]\!]\eta = \mathbb{B}[\![b_{ij}]\!]$

$\mathbb{E}[\![$ if e then e' else e'' $]\!]\eta =$

if $\mathbb{E}[\![e]\!]\eta \in T$ then (if ($\mathbb{E}[\![e]\!]\eta \mid T$) then $\mathbb{E}[\![e']\!]\eta$ else $\mathbb{E}[\![e'']\!]\eta$) else wrong

$\mathbb{E}[\![\{a_1 = e_1, \ldots, a_n = e_n\}]\!]\eta =$

$(\lambda b.$ if $b=a_1$ then $\mathbb{E}[\![e_1]\!]\eta$ else ... if $b=a_n$ then $\mathbb{E}[\![e_n]\!]\eta$ else wrong) in V

$\mathbb{E}[\![e.a]\!]\eta =$ if $\mathbb{E}[\![e]\!]\eta \in R$ then ($\mathbb{E}[\![e]\!]\eta \mid R$)(a) else wrong

$\mathbb{E}[\![[a = e]]\!]\eta =$ <a, $\mathbb{E}[\![e]\!]\eta$> in V

$\mathbb{E}[\![e$ is a $]\!]\eta =$ if $\mathbb{E}[\![e]\!]\eta \in U$ then (fst($\mathbb{E}[\![e]\!]\eta \mid U$) = a) in V else wrong

$\mathbb{E}[\![e$ as a $]\!]\eta =$

if $\mathbb{E}[\![e]\!]\eta \in U$ then (let <b,v> be ($\mathbb{E}[\![e]\!]\eta \mid U$) in if b = a then v else $\perp$) else wrong

$\mathbb{E}[\![\lambda x: \tau . e]\!]\eta = (\lambda v. \mathbb{E}[\![e]\!]\eta\{v/x\})$ in V

$\mathbb{E}[\![e\ e']\!]\eta =$

if $\mathbb{E}[\![e]\!]\eta \in F$ then (if $\mathbb{E}[\![e']\!]\eta \in W$ then wrong else ($\mathbb{E}[\![e]\!]\eta \mid F$)($\mathbb{E}[\![e']\!]\eta$)) else wrong

$\mathbb{E}[\![$ rec x: τ . e $]\!]\eta = \mathbf{Y}(\lambda v. \mathbb{E}[\![e]\!]\eta\{v/x\})$

$\mathbb{E}[\![e: \tau]\!]\eta = \mathbb{E}[\![e]\!]\eta$

Comments on the equations:

- d in V (where $d \in D$ and D is a summand of V) is the injection of d in the appropriate summand of V. Hence d in $V \in V$. This is not to be confused with the let ... be ... in ... notation for local variables.

- $v \varepsilon D$ (where $v \in V$ and D is a summand of V) is a function yielding: $\perp_T$ if $v = \perp_V$; true if $v = d$ in V for some $d \in D$; false otherwise.
- $v \mid D$ (where D is a summand of V) is a function yielding: d if $v = d$ in V for some $d \in D$; $\perp_D$ otherwise.
- if ... then ... else ... is syntax for a function cond: $T \to V \to V \to V$ mapping $\perp_T$ to $\perp_V$.
- equality in L yields $\perp_T$ whenever either argument is $\perp_L$.
- fst extracts the first element of a pair, snd extracts the second one.
- Y is the fixpoint operator of type $(V \to V) \to V$.
- $\mathbb{E}$ defines a call by value semantics, but it allows circular structures to be built.

Intuitively, a well-typed program will never return the wrong value at run-time. For example, consider the 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 $\perp$; the only instance of this in the above semantics is in the equation for e as a.

Having defined $\mathbb{E}$ so that it satisfies the above intuitions about run-time errors, we procede in the following sections by interpreting "e is semantically well-typed" to mean "$\mathbb{E}[\![e]\!]\eta \neq$ wrong", and finally 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 86] $\Im(V)$ (the set of non-empty left-closed subset of V which are closed under least upper bounds of increasing sequences and do not contain wrong). $\Im(V)$ is a lattice of domains, where the ordering is set inclusion. $\Im(V)$ is closed under intersections and finite unions, as well as the usual domain operations.

Here $\mathbb{D} \in TypeExp \to \Im(V)$:

$$\mathbb{D}[\![\iota_i]\!] = B_i \text{ in } V$$
$$\mathbb{D}[\![\{a_i : \tau_i\}]\!] = \bigcap_i \{r \in R \mid r(a_i) \in \mathbb{D}[\![\tau_i]\!]\} \text{ in } V \qquad \text{(where } \mathbb{D}[\![\{\}]\!] = R \text{ in } V\text{)}$$
$$\mathbb{D}[\![[a_i : \tau_i]]\!] = (\{\langle \perp_L , v\rangle \mid v \in V\} \cup \bigcup_i \{\langle a_i , v\rangle \in U \mid v \in \mathbb{D}[\![\tau_i]\!]\}) \text{ in } V$$
$$\mathbb{D}[\![\sigma \to \tau]\!] = \{f \in F \mid v \in \mathbb{D}[\![\sigma]\!] \Rightarrow f(v) \in \mathbb{D}[\![\tau]\!]\} \text{ in } V$$

where D in $V = \{d \text{ in } V \mid d \in D\} \cup \{\perp_V\}$

Theorem ($\mathbb{D}$ properties)

$\forall \tau.\ \mathbb{D}[\![\tau]\!]$ is an ideal (hence $\perp \in \mathbb{D}[\![\tau]\!]$)

$\forall \tau, v.\ v \in \mathbb{D}[\![\tau]\!] \Rightarrow v \neq$ 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.

$\iota_i \le \iota_i$

$\{a_i : \sigma_i , a_j : \sigma_j\} \le \{a_i : \sigma'_i\}$ iff $\sigma_i \le \sigma'_i$ $(i \in 1..n, n \ge 0;\ j \in 1..m, m \ge 0)$

$[a_i : \sigma_i] \le [a_i : \sigma'_i , a_j : \sigma'_j]$ iff $\sigma_i \le \sigma'_i$ $(i \in 1..n, n \ge 0;\ j \in 1..m, m \ge 0)$

$\sigma \to \tau \le \sigma' \to \tau'$ iff $\sigma' \le \sigma$ 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 $\Im(\boldsymbol{V})$. Although this is mathematically appealing, we have chosen not to do so 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 $\Im(\boldsymbol{V})$ model is set inclusion. This allows us to give a very direct semantics to subtyping, as simple set inclusion of domains.

Theorem (Semantic Subtyping)

$\tau \le \tau' \ \Leftrightarrow\ \mathbb{D}[\![\tau]\!] \subseteq \mathbb{D}[\![\tau']\!]$.

The proof is by induction on the structure of τ and τ'. We shall only need the $\Rightarrow$ direction in the sequel.

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 may be shown to be consistent with respect to the semantics of the language, as we shall see at the end of this section.

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 $\leq$ 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 our 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] $$\frac{A \vdash e: \text{bool} \quad A \vdash e': \tau \quad A \vdash e'': \tau}{A \vdash (\text{if } e \text{ then } e' \text{ else } e''): \tau}$$

[RECORD] $$\frac{A \vdash e_1 : \tau_1 \quad \ldots \quad A \vdash e_n : \tau_n}{A \vdash \{a_1 = e_1, \ldots , a_n = e_n\} : \{a_i : \tau_i\}}$$ 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] $$\frac{A \vdash e: [\ldots]}{A \vdash (e \text{ is } a) : \text{bool}}$$

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

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

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

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

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

Some comments on the rules:

- A (called a set of assumptions) is a finite mapping of variables to types; $A(x)$ is the type associated with x in A; $A.x{:}\tau$ is the set of assumptions A extended with the association $x{:}\tau$, i.e. it maps x to τ and any other y to $A(y)$.
- If there are some non-trivial inclusions in the basic types (e.g. $\text{int} \le \text{real}$) then [BAS] must be changed to $A \vdash b_{ij} : \tau$ where $\iota_i \le \tau$.
- 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.
- 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)

$A.x: \sigma \vdash e: \tau$ and $\sigma' \le \sigma \;\Rightarrow\; A.x: \sigma' \vdash e: \tau$.

Theorem (Syntactic Subtyping)

$A \vdash e: \tau$ and $\tau \le \tau' \;\Rightarrow\; A \vdash e: \tau'$.

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 set of assumptions A agrees with an environment η if for all x in the domain of A, $A(x) = \tau$ implies $\eta[\![x]\!] \in \mathbb{D}[\![\tau]\!]$.

Theorem (Semantic Soundness)

if $A \vdash e: \tau$ and A agrees with η then $\mathbb{E}[\![e]\!]\eta \in \mathbb{D}[\![\tau]\!]$.

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

In words, 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 η such that A agrees with η, $\mathbb{E}[\![e]\!]\eta \in \mathbb{D}[\![\tau]\!]$, which implies that $\mathbb{E}[\![e]\!]\eta \neq$ 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. If we have a type expression containing the above operators we can process the expression checking that the operators can indeed be 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 (partial) *meet* operation on types (written $\downarrow$), and or is interpreted as (partial) *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:

$\iota_i \uparrow \iota_i = \iota_i$

$\{a_i : \tau_i , b_j : \sigma_j\} \uparrow \{a_i : \tau'_i, c_k : \rho_k\} = \{a_i : \tau_i \uparrow \tau'_i\}$
if all $\tau_i \uparrow \tau'_i$ are defined $(\forall j,k.\ b_j \neq c_k)$

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

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

$\tau \uparrow \tau'$ undefined otherwise

$\iota_i \downarrow \iota_i = \iota_i$

$\{a_i : \tau_i , b_j : \sigma_j\} \downarrow \{a_i : \tau'_i, c_k : \rho_k\} = \{a_i : \tau_i \downarrow \tau'_i , b_j : \sigma_j , c_k : \rho_k\}$
if all $\tau_i \downarrow \tau'_i$ are defined $(\forall j,k.\ b_j \neq c_k)$

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

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

$\tau \downarrow \tau'$ undefined otherwise

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

τ ignoring a undefined otherwise

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

τ dropping a undefined otherwise

Note that $\uparrow$ may be undefined even if there is a least upper bound with respect to $\leq$ for its operands; similarly for $\downarrow$.

Proposition ($\uparrow$ and $\downarrow$ properties)

If $\sigma \uparrow \tau$ is defined, then it is the smallest ρ (w.r.t. $\leq$) such that $\sigma \leq \rho$ and $\tau \leq \rho$.

If $\sigma \downarrow \tau$ is defined, then it is the largest ρ (w.r.t. $\leq$) such that $\rho \leq \sigma$ and $\rho \leq \tau$.

Let ***S*** be the set of ideals denoted by ordinary type expressions (without $\downarrow$ (and) and $\uparrow$ (or) operators) where r/a = (λb. if b = a then $\perp$ else r(b)).

Proposition

$\mathbb{D}[\![\sigma \text{ and } \tau]\!]$ = the largest ideal in ***S*** contained in $\mathbb{D}[\![\sigma]\!] \cap \mathbb{D}[\![\tau]\!]$	when defined
$\mathbb{D}[\![\tau \text{ ignoring a}]\!]$ = {r $\in$ ***R*** | ((r / a) in ***V***) $\in \mathbb{D}[\![\tau]\!]$} in ***V***	when defined
$\mathbb{D}[\![\sigma \text{ or } \tau]\!]$ = the smallest ideal in ***S*** containing $\mathbb{D}[\![\sigma]\!] \cup \mathbb{D}[\![\tau]\!]$	when defined
$\mathbb{D}[\![\tau \text{ dropping a}]\!]$ = $\mathbb{D}[\![\tau]\!]$ – ({<a, v> $\in$ ***U***} in ***V***)	when defined.

14. Typechecking

The (partial) typechecking function is $\mathbb{T} \in$ ***Exp*** –> ***TypeEnv*** –> ***TypeExp***, where ***Exp*** and ***TypeExp*** are respectively expressions and type expressions according to our grammar, and ***TypeEnv*** = ***Id*** –> ***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 $\mathbb{T}[\![e]\!]\mu = \tau$, we imply that the typechecking of e does not fail.

$\mathbb{T}[\![x]\!]\mu = \mu[\![x]\!]$

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

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

$\mathbb{T}[\![\{a_1 = e_1, \ldots, a_n = e_n\}]\!]\mu = \{a_1 : \mathbb{T}[\![e_1]\!]\mu, \ldots, a_n : \mathbb{T}[\![e_n]\!]\mu\}$

$\mathbb{T}[\![e.a]\!]\mu =$ if $\mathbb{T}[\![e]\!]\mu =$ { ... a: τ ... } then τ else fail

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

$\mathbb{T}[\![e \text{ is } a]\!]\mu =$ if $\mathbb{T}[\![e]\!]\mu =$ [... a: τ ...] then bool else fail

$\mathbb{T}[\![e \text{ as } a]\!]\mu =$ if $\mathbb{T}[\![e]\!]\mu =$ [... a: τ ...] then τ else fail

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

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

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

$\mathbb{T}[\![e: \sigma]\!]\mu =$ if $\mathbb{T}[\![e]\!]\mu = \tau$ and $\tau \leq \sigma$ then σ 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 set of assumptions A if for every x in the domain of A, $\mu[\![x]\!] = A(x)$.

Theorem (Syntactic Soundness)

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

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

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

Corollary (Typechecking prevents type errors):

if $\mathcal{T}[\![e]\!]\mu = \tau$ then $\mathcal{E}[\![e]\!]\eta \neq$ wrong (when $\eta[\![x]\!] \in \mathbb{D}[\![\mu [\![x]\!]]\!]$ for all x).

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

The typechecking algorithm is intentionally more restrictive than the type inference system; it is possible to deduce A.x:bool $\vdash$ if x then {a=true} else {a=3} : {}, but in practice we want this to be a type error for the same reasons that made us rule out the anything type. This restriction is enforced by the definitions of $\uparrow$ and $\downarrow$. Similarly, one can infer any type for [a=3] as b, while the typechecker fails; this is justified since [a=3] as b will always fail at run-time.

For these reasons, we do not have a (perhaps desirable) syntactic completeness theorem of the form: if $A \vdash e: \tau$ and μ agrees with A, then $\mathcal{T}[\![e]\!]\mu$ is defined and $\mathcal{T}[\![e]\!]\mu \leq \tau$. One could strive for syntactic completeness by using the (partial) $\vee$ and $\wedge$ (w.r.t. $\leq$) instead of $\uparrow$ and $\downarrow$ in the typechecking algorithm (then the modified algorithm computes $\mathcal{T}[\![$if x then {a=true} else {a=3}$]\!]\mu$ = {}), and by replacing the is and as primitives by a case construct.

15. Conclusions

This work originated as an attempt to justify the multiple inheritance constructs present in the Galileo data base language [Albano 85] and to provide a sound typechecking algorithm for that language. The Amber language [Cardelli 86] 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 with 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 are addressed in [Cardelli 85].

There are now several competing (although not totally independent) styles of parametric polymorphism, noticeably in [Milner 78], [Reynolds 74, McCracken 84] and [MacQueen 86]. Inheritance is orthogonal to all of these, so it seems better to study it independently, at least initially. However, the final goal is to achieve full integration of parametric polymorphism and multiple inheritance, merging functional programming with object-oriented programming at the semantic and typing levels; this problem is currently receiving much attention.

16. Related work and acknowledgements

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

Finally, I would like to thank David MacQueen for many discussions, John Reynolds and the referees for detailed suggestions and corrections, and Antonio Albano and Renzo Orsini for motivating me to carry out this work.

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 85] A.Albano, L.Cardelli, R.Orsini: **Galileo: a strongly typed, interactive conceptual language**, IEEE Transactions on Database Systems, June 1985.

[Attardi 81] G.Attardi, M.Simi: **Semantics of inheritance and attributions in the description system Omega**, M.I.T. A.I. Memo 642, August 1981.

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

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

[Cardelli 86] L.Cardelli: **Amber**, *Combinators and Functional Programming Languages, Proc. of the 13th Summer School of the LITP*, Le Val D'Ajol, Vosges (France), May 1985. Lecture Notes in Computer Science n. 242, Springer-Verlag, 1986.

[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**, in *Semantics of Data Types*, Lecture Notes in Computer Science n.173, Springer-Verlag 1984.

[MacQueen 86] D.B.MacQueen, G.D.Plotkin, R.Sethi: **An ideal model for recursive polymorphic types**, Information and Control 71, pp. 95-130, 1986.

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

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

[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: **Objects, Message Passing, and Flavors**, chapter 20 of *Lisp machine manual*, Fourth Edition, Symbolics Inc., 1981.