Safe Run-time Overloading

François Rouaix
INRIA*
B.P.105 78153 Le Chesnay Cedex, France

Abstract

We present a functional language featuring a form of dynamic overloading akin to message passing in object oriented languages. We give a dynamic semantics describing a non-deterministic evaluation, as well as a type discipline (static semantics) supporting type inference. The type system ensures that a well-typed program has a correct execution, unique up to a semantic equivalence relation, and allows this execution to proceed deterministically, while resolving overloading at run-time.

1 Presentation

1.1 Motivation

When analyzing the history of relations between language design and types, we can point out two different uses of types in a language. The first one has for purpose the guarantee of correct execution of a well-typed program. A static type-checker ensures an execution free of type-errors. This concept is mainly popular in languages designed as formal systems, such as ML. The second one is to help gaining efficiency, and design facility for programs. Using type information, one may be able to compile some features of the language with optimizations, or to provide tools for program maintenance [SchaCoo 86, MeyNM 87, Johnson 86]. While formal systems need new constructs (such as abstraction from implementation), object-oriented languages need formalization and types. We present a system which incorporates concepts from the two worlds.

*This work has been partially supported by the Eureka Software Factory (ESF) project

1.2 Intuitive Overview

One of the ideas underlying the system presented here is that these two points of view (formal types vs. efficient implementation) are to be treated separately, following ideas in CLU [Liskov 81], Emerald [RajLev 89] or Duo-Talk [Lunau 89]. If we take inheritance, we think that it should be split in two distinct issues: the first aspect is property inheritance (or behavior inheritance), which says that, for example,"a child has more properties than its parent". It should concern only the type-system. The second aspect is sharing data representation or primitive functions between child and parent, and this is an implementation issue.

In ML's interpretation of types, a functional value is said polymorphic if it has many types. Now, reversing this definition, one may understand a polymorphic function (or object such as nil) as a description of a set of monomorphic values (one for each instance) sharing the same implementation. The idea here is to extend the notion of parametric polymorphism by qualifying type variables with sets of properties. These properties will be used to constrain the set of permitted implementations for the entity having a "qualified polymorphic type". Although this extension is similar to *ad-hoc* polymorphism [CarWeg 86, WadBlo 89], it behaves like parametric polymorphism. In particular, function types are covariant with argument types, with respect to our definition of type inclusion.

We introduce a restricted form of overloading, that can be typed with usual polymorphism, and we argue that it addresses a wide range of programming issues, such as object-oriented languages or code reuse.

Let us describe what kind of overloading mechanism we want for our language.

1. Resolution may be done at run-time. That is, the value denoted by an overloaded symbol is not necessarily determined statically from the program.

2. Resolution of overloading does not fail at runtime. The (correct) solutions are the values that

will not cause a type-error.

3. When multiple solutions are found, all possible solutions are "equivalent".

The idea is to delay the resolution of an overloaded symbol to its occurrence where the context provides enough information to choose a safe value. This mechanism is very close to the "call by name" of Algol 60. Like call by name, the suspension may yield different values at each evaluation. However, here the changes are due to contextual typing constraints instead of side-effects since our language is purely applicative. The syntactic construct in the language that introduces this form of evaluation is the *let in* construct. If we take the example

$$let\ double = \lambda x.(x + x)\ in\ (double\ 3, double\ \pi)$$

where $+$ is overloaded with addition of *Integer* and addition of *Float*, we see intuitively that lazy expansion for *double* allows a safe choice for both occurrences of $+$.

Informally, we consider a set of specifications (i.e. types) of overloaded symbols, a set of specifications of implementations (resolution values of overloaded symbols), and want to determine if the code we write is compatible with these specifications. This is achieved by interpreting types of objects as sets of properties, sometimes called classes.

The typing may be found very flexible when comparing with other systems. The way the typing is designed makes that highly overloaded code is almost always typable (the only errors detected are applications of non-functional objects). It just means that the code may be executed provided there exists a proper implementation of the primitives that are used. In a way, the type inference computes the minimal constraints on the code, imposed by some specifications, such that fulfilling these constraints ensure a proper execution. The advantage of minimal constraints is that they give maximum re-use.

In terms of object-oriented programming, overloaded symbols are methods, but more powerful, since the selection is not necessarily on the first argument, and they are first-class citizens of the functional language (which is also the choice in CLOS [DeMGab 87]). Polymorphic functions or objects may be seen as methods associated to meta-classes[1].

[1] Meta-classes are entities for grouping common method definitions among different classes

2 Comparison with current solutions

We are interested in existing solutions for functional languages implementing run-time overloading and type inference.

Kaes work

The formalism presented in [Kaes 88] has the same basis: qualified type variables and overloading schemes. The enhancements presented here are mainly: dynamic semantics for an *untyped* functional language featuring overloading, a known framework for operations on types, and more general overloading schemes.

Haskell

Haskell proposes a class system with similar goals. However, the approach is different on some points. We will go into certain details because they enlighten some aspects of our system.

- The first essential difference is that Haskell imposes the declaration of classes. This strongly influences the type-inference by forcing computed types to belong to a predefined hierarchy of types. This is an argument for readability of types and strictness of programming, but tends to be a drawback in the areas of object-oriented programming and reusability. For example, modifying the hierarchy of classes (while keeping the same definitions of overloaded symbols) greatly changes the typing. The advantage we have here is that we compute types in a larger set with a more convenient structure. There are cases where Haskell would produce a type-error because it tries to infer a class which is not predeclared. Our choice is that any inferred "class" is legitimate. The matter that there exists an implementation for this "class" may or may not be relevant depending on the context. If it is relevant, then the user-interface may inform the user of what actual implementations exists and which primitive they lack to match the "class".

- A second important difference is the notion of principal type. In Haskell, static resolution when ambiguity arises is considered as a failure because there is no principal types. In our point of view, the principal type still exists before static resolution. Static resolution consists in an arbitrary choice, but all choices have to be equivalent from a semantic point a view. We certainly do not have a principal type property after static resolution, but we don't consider this as a failure. The point here is that in our system, we may not be able to translate a well-typed term into an ML term by following Haskell technique ([HudWad 88],p. 64).

SML

SML provides another form of overloading, that should be solved statically. Expressions such as $\lambda x.(x + x)$ where $+$ is overloaded are not typable. However, in the spirit of giving tools for program maintenance, SML provides modules. These modules also provide a form of overloading, that must be solved with aid of the user, using explicit qualifiers to select primitives from a structure previously opened. If we consider only this aspect of modules, then we may say by analogy that we infer the minimal "open" statements (we infer the signature of the module as well), and the qualifiers are now implicit[2]. The same issue of declaring or not structures appears here.

Work on OO languages

There has been considerable work in the OO community to design type systems for safe object-oriented languages. The mechanisms involved are delegation and inheritance (from the language approach), subtyping and subclassing (from the type approach). The system presented here relies mainly on a covariant subclass relation, coded by polymorphism. The separation of the hierarchies allows a separate treatment for the subtyping hierarchy.

3 Formal Language definition

To formalize these concepts, we define a simple language, its dynamic and static semantics and then show how they interact[3]. This language has only the minimum constructions to express that it is functional and has overloading. Detailed examples are given in an appendix.

If Id is a denumerable set of non-overloaded symbols and Oid a finite set of overloaded symbols, we note e an expression ranging over $OExp$ defined by:

$$\begin{array}{rcll} e & := & b & \text{basic constant} \\ & | & x & \text{where } x \in Id \\ & | & o & \text{where } o \in Oid \\ & | & \lambda x.e & \text{abstraction} \\ & | & e\,e' & \text{application} \\ & | & let\ x = e\ in\ e' & \text{lazy expansion} \end{array}$$

3.1 Dynamic Semantics

The semantical objects of the langage are:

- basic values (constants) in predefined sets $B_1, \ldots, B_n$ and built-in primitives (closures)
- closures, composed of a symbol name (formal parameter), a body expression and an environment, for λ-abstractions.
- thunks, composed of an expression and an environment, for *let* expressions.

There are two kinds of environments, E used in closures and thunks and a global environment O, which describes the values associated to each overloaded symbol. Formally,

$$\begin{array}{rcl} b & \in & BaseValues = B_1 \cup \ldots \cup B_n \\ {[x, e, E]} & \in & Closures = Id \times OExp \times Env \\ \langle e, E\rangle & \in & Thunks = OExp \times Env \\ v & \in & Val = BaseValues + Closures \\ E & \in & Env = Id \rightarrow (Val + Thunks) \\ O & \in & OEnv = Oid \rightarrow \mathcal{P}(Val) \end{array}$$

The dynamic semantics (Figure 1) is a set of inference rules, describing evaluation of expressions in $OExp$. Sequents are of the form $E \vdash_d e \longrightarrow v$, meaning that v can be derived from e in environment E (O is global).

An evaluation is a proof in this system. This semantics is clearly non-deterministic, since the evaluation of an overloaded symbol may yield any of its possible values (cf. rule OID_d). There is no rule defining a wrong execution, nor a specific *wrong* value. Failure is expressed by the fact that no rule is available to interpret an expression or by type-errors in application of built-in closures. Besides non-determinism, another important point is the interpretation of the *let* construct. The object $\langle e, E\rangle$ *freezes* the source code just like in a closure. The purpose is to delay the actual evaluation of the binding value until it is really used, possibly with different evaluations for each occurrence. The ID_d rule describes this evaluation, by calling a small subsystem $REAL_{d'}$ and $TAUT_{d'}$

[2]provided primitives with the same name in different structures are defined homogeneously

[3]All semantics are written with Natural Semantics [CleDDK 86], this choice being justified later

Figure 1: Dynamic semantics

$$OID_d \qquad \frac{o \in Dom(O) \qquad v \in O(o)}{E \vdash_d o \longrightarrow v}$$

$$ID_d \qquad \frac{x \in Dom(E) \qquad \vdash_{d'} E(x) \longrightarrow v}{E \vdash_d x \longrightarrow v}$$

$$REAL_{d'} \qquad \frac{E \vdash_d e \longrightarrow v}{\vdash_{d'} \langle e, E\rangle \longrightarrow v}$$

$$TAUT_{d'} \qquad \frac{}{\vdash_{d'} v \longrightarrow v}$$

$$ABS_d \qquad \frac{}{E \vdash_d \lambda x.e \longrightarrow [x, e, E]}$$

$$APP_d \qquad \frac{E \vdash_d e \longrightarrow [x'', e'', E''] \qquad E \vdash_d e' \longrightarrow v' \qquad E'' \cup \{x'' \mapsto v'\} \vdash_d e'' \longrightarrow v}{E \vdash_d e\, e' \longrightarrow v}$$

$$LET_d \qquad \frac{E \cup \{x \mapsto \langle e, E\rangle\} \vdash_d e' \longrightarrow v'}{E \vdash_d let\ x = e\ in\ e' \longrightarrow v'}$$

which says how to *realize* a *thunk*. Strict deterministic evaluation would force the resolution of overloading statically. Non-determinism was the significant argument for choosing Natural Semantics over Denotational Semantics. It is totally implicit in Natural Semantics.

3.2 Correct programs

As a program (a closed expression of $OExp$) may have different executions, we need a notion of *correctness* for program execution.

Definition 3.1 *Let S be some equivalence relation on the set Val of values that we call semantic equivalence. Execution of a term $e \in OExp$ is* correct *for S iff*

- *there is a proof of $\vdash_d e \longrightarrow v$ in $\vdash_d$, $\vdash_{d'}$ for some v.*
- *all such v are equivalent in S*

These definitions do not imply the use of a type-system. Naturally this is a formal reference semantics, and the typing algorithm will allow us to transform the original term in a term that can be executed deterministically according to one of the safe computations.

3.3 Static semantics

Our goal is to provide an algorithm that, given a term in $OExp$ and an overloading environment, will determine statically if there exists a correct execution in the former system. For this purpose, we will use a type-system derived from ML. If we examine our requirements on overloading, we reasonably choose that

- the type expression for an overloaded symbol in the environment should be polymorphic.
- the actual value of an occurrence of an overloaded symbol depends on its type at this occurrence, this type reflecting somehow the context of application of this overloaded symbol. We then need a new syntactic form for types of overloaded symbols.
- This new syntactic form should have some properties we can use when defining our semantic equivalence

An overloaded symbol acts like a polymorphic object, the values of which are chosen from its type instance when it is used. This interpretation extends to polymorphic functions, which are now abstract functions, "realized" when they are used.

We have chosen a language of sorted and rational terms[4] for the types. The following consists in its definition and many useful notations. Most of this is borrowed from Rémy's extension to ML types for records and variants.

Expressions are of two sorts *Type* and *Field*. Signatures of function symbols are written as usual with $\Rightarrow$ and $\otimes$ operators. Given

- B a set of basic constants of sort *Type*
- L a finite ordered set of symbols[5] of cardinality l.
- the constructors $\mathcal{C} = \{\rightarrow, \Delta, \nabla, \Theta\}$ with signatures:

$$\begin{array}{lll} \rightarrow & :: & Type \otimes Type \Rightarrow Type \\ \Delta & :: & Type \Rightarrow Field \\ \nabla & :: & Field \\ \Theta & :: & \underbrace{Field \otimes Field \otimes \ldots Field}_{l} \Rightarrow Type \end{array}$$

- two denumerable sets of variables $\mathcal{V}^t, \mathcal{V}^f$ of respective sorts *Type, Field*, their union is denoted by $\mathcal{V} = \mathcal{V}^t \cup \mathcal{V}^f$
- two denumerable sets of generic variables $\mathcal{V}_g^t, \mathcal{V}_g^f$ of respective sorts *Type, Field*, their union is denoted by $\mathcal{V}_g = \mathcal{V}_g^t \cup \mathcal{V}_g^f$,

we define

- $\mathcal{R}$ as the set of first order sorted regular trees constructed over the set of variables $\mathcal{V}$, constants in B and constructors in $\mathcal{C}$. Elements of R of sort *Type* are called *types*. Their subset is noted $\mathcal{T}$.
- $\mathcal{R}_g$ as the set of first order sorted regular trees constructed over the set of variables $\mathcal{V}_g \cup \mathcal{V}$, constants in B and constructors in $\mathcal{C}$. Elements of $\mathcal{R}_g$ of sort *Type* are called *generic types*. Their subset is noted $\mathcal{T}_g$.
- A (non-generic) *graft*[6] μ is a mapping from $\mathcal{V}$ to $\mathcal{R}$ respecting the sorts.
- A *total* graft is a *total* mapping from $\mathcal{V}$ to $\mathcal{R}$ respecting the sorts.
- A *generic graft* μ_g is a mapping from $\mathcal{V}_g$ to $\mathcal{R}_g$ respecting the sorts.
- A *monotype* (or ground type) is a type with no variables (noted ι).
- A graft is *ground* if it ranges in monotypes.

In the following we use the notations: α (resp. α_g) for type (resp. generic type) variables ; σ, τ for types; σ_g, τ_g for generic types ; ϕ for field variables (i.e. variables of sort Field). We will note $\tau\mu$ (resp. $A\mu$) the application of the graft μ to τ (resp. A).

To help understanding what follows, here is an intuitive interpretation of these new constructions. A non-functional type is denoted by a Θ-term. The fields correspond to the possible properties (e.g. primitive functions) of the type. The construction $\Delta(\tau)$ means that the field is present with the type τ. The construction ∇ means that the field is absent. An *abstract data type* (ADT) may be seen as an open set of properties. "Open" means that this set may be extended during unification for example. Such an extension restricts the possible implementations of this ADT.

In our formalism, an ADT is a type constructed with Θ such that all fields are either variables or constructed with Δ, (not all being Δ since ADTs are open)[7].

An *implementation* is a type constructed with Θ with no field variable. It is a non-expandable set of properties. We will be using the name of the implementation as a special label because we must be able to tell between two implementations of a same ADT. This label also acts as a discriminant when unifying two implementations.

Definition 3.2 (Abstract,Real) *By extension, we will say that a type σ_g is* abstract *if at least one subterm of σ_g is an ADT. Otherwise σ_g is said to be* real.

3.4 Overloading scheme and implementations

This section shows how we are going to use these new type constructs. The overloading environment is a set of overloading schemes, which intuitively, describe the type of an overloaded symbol, together with a set of implementation descriptions.

For any set V of symbols, we define the language $Simple(V)$ by

$$\begin{array}{rcl} s & := & v \in V \\ & | & s \rightarrow s \end{array}$$

[4]Intuitively, a rational term may be seen as an infinite term with a regular structure.

[5]These symbols in L are intended for field names in Θ-terms (see below). Since each Θ-term has a field for each of the symbols in L, they are left implicit, and the correspondance is based on field order and L order

[6]In non-rational terms, a graft is a substitution

[7]see appendix for examples

Figure 2: Examples of overloading schemes

$$
\begin{array}{lll}
+ & \leadsto & (\omega \rightarrow \omega \rightarrow \omega, \{\}) \\
fst & \leadsto & (\omega \rightarrow \alpha_g, \{\ left \ : \ \alpha_g\ \}) \\
snd & \leadsto & (\omega \rightarrow \alpha_g, \{\ right \ : \ \alpha_g\ \}) \\
hd & \leadsto & (\omega \rightarrow \alpha_g, \left\{ \begin{array}{lcl} elem & : & \alpha_g \\ tl & : & \omega \rightarrow \omega \\ cons & : & \alpha_g \rightarrow \omega \rightarrow \omega \end{array} \right\}) \\
cons & \leadsto & (\alpha_g \rightarrow \omega \rightarrow \omega, \left\{ \begin{array}{lcl} elem & : & \alpha_g \\ tl & : & \omega \rightarrow \omega \\ hd & : & \omega \rightarrow \alpha_g \end{array} \right\}) \\
nil & \leadsto & (\omega, \{\})
\end{array}
$$

$$
\begin{array}{l}
Oid = \{+, fst, snd, hd, cons, nil\} \\
Qualifiers = \{left, right, elem\} \\
\alpha_g \in \mathcal{V}_g^t
\end{array}
$$

Let $Qualifiers$ be a finite set of symbols, and ω be a special symbol[8]. Let $\mathcal{I}$ be a set of symbols to be used as implementation names.

Definition 3.3 (Implementation description) *An implementation description is a binding* $i \sim \rho$ *where*

- $i \in \mathcal{I}$ *is the implementation name.*
- ρ *is a mapping from* $Qualifers \cup Oid$ *to* $Simple(\mathcal{V}_g^t \cup \{\omega\} \cup \{B_1, \ldots, B_n\} \cup (\mathcal{I} - i))$

ρ describes what primitives the implementation provides, and eventually what properties it has. A minimal implementation of natural numbers could be described by

$$
num_0 \sim \left\{ \begin{array}{lcl} natural & : & \omega \\ 0 & : & \omega \\ succ & : & \omega \rightarrow \omega \end{array} \right\}
$$

An implementation description $i \sim \rho$ is expanded in a Θ-type, built from ρ, using the following rules:

- the set L is defined as $Qualifiers \cup \mathcal{I} \cup Oid$
- ω denotes an occurrence of the Θ-type we are building (i.e. it expresses recursion in the type).
- the fields of the Θ-type are $\Delta(\rho(q))$ when q is defined in ρ, $\Delta(\omega)$ for the field corresponding to i, and ∇ for other fields.

As expected, the generated type is an *implementation*, according to our terminology. It has a field corresponding to its name, to be used as a discriminant with other implementations with the same primitives and properties.

Definition 3.4 (Overloading scheme) *An overloading scheme is a binding* $o \leadsto (s, \rho)$ *where*

- $o \in Oid$
- $s \in Simple(\mathcal{V}_g^t \cup \{\omega\} \cup \{B_1, \ldots, B_n\} \cup \mathcal{I})$ *such that* ω *appears in* s
- ρ *is a mapping from* $Qualifiers \cup Oid$ *to* $Simple(\mathcal{V}_g^t \cup \{\omega\} \cup \{B_1, \ldots, B_n\} \cup \mathcal{I})$*, such that* $\rho(o) = s$.[9]

Examples of overloading schemes are given in Figure 2.

An overloading scheme $o \leadsto (s, \rho)$ is expanded in a type, built from s, using the following rules:

- the set L is defined as $Qualifiers \cup \mathcal{I} \cup Oid$
- ω denotes an occurrence of an Θ-type which fields are built using ρ: the fields are $\Delta(\rho(q))$ when q is defined in ρ, $\Delta(s)$ for the field corresponding to o, and a fresh generic field variable $\phi_g \in \mathcal{V}_g^f$ for each other field.

Intuitively, ω can be considered as a type variable which instantiation is restricted by the presence of properties described in ρ. Polymorphism appears in the field variables ϕ_g which provide the possibility of later instantiation that changes the status of the Θ type from an ADT to an implementation. The recursion on ω (i.e. the presence of $o : s$ in the Θ construction, implicit in the figure examples) shows where we will take the actual value of o: when the Θ-type becomes an implementation, then this implementation will provide the specific adequate value for o in its primitives. One can read the example hd in Figure 2 as $hd : \theta \rightarrow \alpha$, such that θ has properties $hd : \theta \rightarrow \alpha$, and also $elem : \alpha$, $tl : \theta \rightarrow \theta$, $cons : \alpha \rightarrow \theta \rightarrow \theta$.

Although the description of an implementation is syntactically similar to an overloading scheme, it is expanded into a "closed tree", rather than an "open" tree (i.e. ADT) for overloading schemes.

[8]one may think of ω denoting *self* in object-oriented languages

[9]For conciseness, the value $\rho(o)$ is not given in the description of ρ

Figure 3: Typing rules

$$OID_s \qquad \frac{o \in Oid \qquad \sigma \in \lfloor A(o) \rfloor}{A \vdash_s o : \sigma}$$

$$ID_s \qquad \frac{x \in Id \qquad \sigma \in \lfloor A(x) \rfloor}{A \vdash_s x : \sigma}$$

$$ABS_s \qquad \frac{A_x \cup \{x \mapsto \sigma\} \vdash_s e : \tau}{A \vdash_s \lambda x.e : \sigma \to \tau}$$

$$APP_s \qquad \frac{A \vdash_s e : \sigma \to \tau \qquad A \vdash_s e' : \sigma}{A \vdash_s ee' : \tau}$$

$$LET_s \qquad \frac{A \vdash_s e : \sigma \qquad A_x \cup \{x \mapsto \lceil A, \sigma \rceil\} \vdash_s e' : \tau}{A \vdash_s let\, x = e\, in\, e' : \tau}$$

3.5 Typing rules

Static semantic rules use sequents of the form

$$A \vdash_s e : \tau$$

where :

$$\begin{array}{rcl} \tau, \sigma & \in & \mathcal{T} \\ \sigma_g & \in & \mathcal{T}_g \\ A & \in & TEnv = (Id \cup Oid) \to \mathcal{T}_g \end{array}$$

Following [CleDDK 86, Rémy 89], we define a modified version of the original typing rules [Milner 78, DamMil 82] in Figure 3, where[10]

- $\lfloor \sigma_g \rfloor$ is the set of instances of σ_g, i.e. types obtained from σ_g by grafting all generic variables in σ_g by trees in $\mathcal{R}$
- Generalization $\lceil A, \sigma \rceil$ is the grafting of all non-generic variables in σ which do not appear in A by new generic variables.
- A_x denotes A where the binding $x \mapsto A(x)$ has been removed if it exists.

Remark 3.1 *Since Id and Oid are disjoint, the mapping $A \downarrow Oid$ [11] is never modified in $\vdash_s$.*

As proved in [Rémy 89], there exists an algorithm (W), sound and complete, that computes a principal type for expressions in *OExp*. Basically, we are re-using here Rémy's result that the original W algorithm designed by Milner may be extended to rational types, with Huet's unification algorithm[Huet 76]. Types are also simpler than in Rémy's system since we don't consider records as objects of the language, so that operations on types do not introduce the problem of forgetting fields. Note that this part of the type-checking is really syntactic work on the language. Our special interpretation of the *let in* construct does not change the typing. Also, since recursive types in W allow the typing of expressions such as $\lambda x.(x\ x)$ or the Y combinator, we must be careful when interpreting types. In this presentation, we forbid recursive types which are not Θ-types.

It should be noted that the algorithm W, as well as computing a principal type[12] for an expression, annotates (deterministically) each node of the expression by its type. We remind here an essential property of the static semantics:

Lemma 3.1 (Sub) $A \vdash_s e : \tau \Rightarrow A\mu \vdash_s e : \tau\mu$ *(where μ is a non-generic graft.)*

Remark 3.2 *In the inference system defined above,* generic types *appear only in type-environments A.*

3.6 Safety and static resolution

We now need to prove the consistency of static and dynamic semantics, which is that a well-typed term has a correct execution. The first point is that the type computed by W is not sufficient to determine

[10]the rule for basic constants is omitted, since we can treat them as identifiers in *Id*

[11]$A \downarrow Oid$ (resp. $A \downarrow Id$) denotes the restriction of A to to *Oid* (resp. *Id*).

[12]which, in this presentation, is a type, not a generic type as noted by Tofte[Tofte 87]

if there exists an execution. Using the overloading environment described in Figure 2, the term

$$\lambda x.\ hd(cons\ x\ nil)$$

would be typed by $\alpha \rightarrow \alpha$. Typing does not guarantee that we have compatible values for hd, cons and nil. The problem comes from variables of sort *Field* that are created by ID_s or OID_s, and eliminated during APP_s rule. These *Field* variables may be thought of as variables that delay the choice of an implementation.

Definition 3.5 (Safety) We need a caracterization of types and grafts that takes in account the interpretation of *Field* variables.

- *A type* σ_g *is* safe *if either* σ_g *is* real *or all field variables in* σ_g *are generic.*
- *A type environment A is safe if* $\forall x \in A$, $A(x)$ *is safe.*
- *A graft* μ *is* safe *for* τ *if* $\tau\mu$ *is safe*
- *A graft* μ *is safe for A if* $A\mu$ *is safe*

Static Resolution: We define here an algorithm which purpose is to complete W to get the consistency, while keeping some overloading resolution at run-time. We consider an expression e decorated by τ , (noted e :: τ), where τ is the type inferred by W (i.e. the principal type τ such that $A \vdash_s e : \tau$), and where each subexpression of e is decorated in the same fashion. Let-bindings[13] are decorated with their generalized types rather than the simple type during this algorithm. Let *OFree* be the set of *non-generic field variables* in all types decorating e. Find a ground graft *SR* on *OFree*, safe for *A* and all types in $e :: \tau$. If we can't find *SR* then the typing fails. If we find *SR*, then *SR* is applied on the proof tree of *W* in the following, and we note the sequents in the proof $A \vdash_s^{W+SR} e : \tau$.

Lemma 3.2 (Static resolution) *All types in* $(e :: \tau)SR$ *are safe. Sequents* $A \vdash_s^{W+SR} e : \tau$ *with* τ *abstract only appear in the proof under application of the* LET_s *rule.*

Proof: by construction, since the *SR* removes all non-generic field variables. Abstract types are now necessarily generic.
Example:(following previous specifications of overloaded symbols) $\lambda x.hd(cons\ x\ nil)$ is typed with

$$\alpha \rightarrow \alpha$$

[13]i.e. the e expression in *let* $x = e$ *in* e'

with SR forcing the choice of compatible values for hd, cons and nil.

Finding the ground graft *SR* implies an arbitrary choice when there are multiple implementations for a same ADT. We must remind this when we will examine the relation between static resolution and our definition of correct execution.

Another remark concerns the implications of static resolution on a real implementation of this system in a language. If we consider a closed term (i.e. a term as defined in the formalism *OExp*), then static resolution may be seen as a two-step operation, which consists in macro-expansion[14] of *let* expressions, and then full resolution of overloaded symbols. However, in a toplevel loop for example, which consists in "open lets", such as *let* x = e;; (expression that does *not* belong to OExp), there is no macro-expansion. As said before, the identification of overloaded symbols in e will be delayed until the typing of an occurrence of x provides enough information. The purpose of *SR* in this case is to identify overloaded symbols that do not depend on future instantiations of the type of x, and to solve them immediately.

3.7 Consistency

The object of this section is to give the semantic interpretation of types in our system. This interpretation[15] is given by a relation $\models$ between $v \in Val + Thunks$ and $\sigma_g \in \mathcal{T}_g$.

We first define the relation on *BaseValues* and monotypes. Note that *implementations* are not necessarily monotypes. They may contain generic variables of sort *Type*. The point is that implementations types do not contain *field* variables.

- $\models b : \iota_B \in \{B_1, \ldots, B_n\}$ by definition of the built-in environment
- $\models v : \iota_I \in \mathcal{I}$ where ι_I is a ground type, by definition of the built-in environment
- $\models [x, e, E] : \iota_1 \rightarrow \iota_2$ if for all v such that $\models v : \iota_1$, there exists r such that
  - $E \cup \{x \mapsto v\} \vdash_d e \longrightarrow r$
  - $\models r : \iota_2$

A type environment A is said *closed* and noted $\overline{A}$ when all variables (of sort *Field* or *Type*) in types on the range of $\overline{A}$ are generic.

[14]in fact redex eliminitation, with eventual renamings
[15]based on [Tofte 87]

The semantic relation $\models$ is extended on types, generic types and environments. We need a definition for environment with non-generic variables (i.e. not closed) for the proofs.

- $\models v : \sigma$ if for all total, ground, safe graft μ, $\models v : \sigma\mu$
- $\models v : \sigma_g$ if for all safe generic grafts μ_g, $\models v : \sigma_g\mu_g$
- $\models \langle e, E\rangle : \sigma_g$ if for all safe instance σ_0, of σ_g there exists v such that $\vdash_{d'} \langle e, E\rangle \longrightarrow v$ and $\models v : \sigma_0$,
- $\models O : \overline{A} \downarrow Oid$, if $\forall o \in O$, $\forall v \in \underline{O}(o)$, (with $\models v : \sigma_g$), σ_g is real and $\exists \mu_g, \sigma_g = \overline{A}(o)\mu_g$
- $\models E : \overline{A} \downarrow Id$ if $\forall x \in Dom(E)$, $\models E(x) : \overline{A}(x)$
- $\models E : \overline{A}$ if $Dom(E) \cup Dom(O) = Dom(\overline{A})$, $\models O : \overline{A} \downarrow Oid$, $\models E : \overline{A} \downarrow Id$.
- $\models E : A$ if for all total, ground, safe graft μ, $\models E : A\mu$

Theorem 3.1 (Consistency) *If* $\models E : A$, $A \vdash_s^{W+SR} e : \tau$ *, and* A *is safe, then*

$$\exists r, E \vdash_d e \longrightarrow r, \text{ such that } \models r : \tau$$

Sketch of Proof: by structural induction on e. The game consists in using the premises of static rules to prove (by induction) the premises of the dynamic rules, thus finding a safe derivation [16]. Note that when A is not safe, the theorem does not hold (non-generic field variables in types potentially means unresolved overloading with possible failure). We need the following lemma:

Lemma 3.3 (Realization) *If* $\models E : A$ *and* $A \vdash_s^{W+SR} e : \tau$ *then for all total, ground, safe graft* μ,

$$\exists r, E \vdash_d e \longrightarrow r, \text{ such that } \models r : \tau\mu$$

This lemma does not give unicity of the result r. However, we may now write the proof of the Consistency theorem, using the static resolution lemma. All inductions but for let-bound symbols are done on the theorem itself, and for these symbols we use the Realization lemma. The important point is that during induction on the theorem itself, A and types are always safe.

[16]For a given term structure in *OExp*, there is only one possible rule for static semantics and for dynamic semantics

3.8 Correctness

Static resolution may involve an arbitrary choice of implementations to transform the non-deterministic program in a deterministic one. We need to be sure that this choice is compatible with our correctness definition. Intuitively, the typing inferred by W gives the observable behaviour of the values derived in the program. The minimum semantic equivalence between different values derivable from the same term is defined by the set of operations[17] that can be applied to these values. The type inferred by W is exactly this set of properties. SR as defined above is then seen as the choice of a set of implementations satisfying these properties. Therefore, any valid choice from SR is compatible with the equivalence relation defined by the type inferred in W.

4 Recursion

We already said that we considered recursive types as invalid (except those cycling on Θ). Static semantics supports them (syntactically) but we have no interpretation of these types in the $\models$ relation. We still can add recursion in *OExp*, using *rec* rules in static and dynamic semantics to provide the same kind of extension as for ML, with the Y combinator for example. The type of Y is simply declared in the built-in environment. From a practical point of view, it is easy to add recursion both in the language and in the typing. We believe that the definition of correctness of execution with recursion could be a step-by-step equivalence of possible executions.

Note also that we don't have "polymorphic recursion". We then know that overloading resolution will be the same in each recursive call, since the resolution happens at instantiation time.

5 A real language

The previous sections deal only with the formal presentation of the language. It is clear that a real implementation is not a non-deterministic language, and that we do not want to re-compute completely thunks when they are used. However, we are able to use the information gathered during type-inference to produce a compiled code for thunks. The occurrence of overloaded symbols that depend on the type of the instance may be replaced by a look-up function call to their corresponding property in the type. This means that the evaluation of a thunk needs a new argument

[17]possibly extended with properties

which is the type of its instance. This new evaluation mechanism becomes deterministic, with respect to the arbitrary choices during static resolution. Note that tagging values with their implementation type does not help, since we may have to select a value from the result type.

The construction above neither describes how implementations are managed, nor details the static resolution. As noted in the introduction, this issue belongs to a different topic. Some issues are:

- when dealing with polymorphic implementations during static resolution, it should be noted that only field variables are grafted. The implementation is *not* an instance of the unresolved type.
- relying in structural equivalence of implementation types as semantic equivalence if not sufficient. However, as our types are compatible with record types, it seems natural to use formalisms such as [CooHC 90] to code implementations with records. Stronger semantic relations may be enforced by direct operations on the primitives of implementations.

In fact, the whole hierarchy dealing with code inheritance, or delegation, has to be managed in the implementation's world.

There are still some major limitations in our approach, due to the very simplicity of the instantiation relation considered as a "subclass" relation. It is not possible to have more than one occurrence of a primitive symbol in a Θ type. This makes impossible to overload, for example, matrix multiplications by a matrix and by a scalar and use them simultaneously in a program. However, this kind of overloading is more a syntactical facility than a conceptual notion of abstraction or code reuse.

Conclusion

We presented in this paper a new approach for overloading that can reconciliate functional programming and object-oriented concepts (abstraction, subclassing, overloading) using familiar techniques such as call-by-name and polymorphism, while keeping useful properties of strong static typing. The formalism presented here is not only an extension of existing type-checkers but rather an attempt for a full system with adequate semantics. The definition of overloading schemes being rather flexible, a wide range of programming methodologies may be modelled by this system. We believe that this formalism can be turned into a real language, with more work on implementation management and new constructs for manipulating the overloading environment. Future work includes also generalization of overloading schemes and run-time extensions of the overloading environment.

Acknowledgements. I am grateful to B. Lang for initiating this work as a formalisation of the implicit typing used in the VTP [Lang 86], and for his support. D. Rémy explained to me many key points in ML typing. B. Lang, V. Donzeau-Gouge and J.-J. Lévy helped in choosing the dynamic semantics.

References

[BorIng 82] A.H. Borning, D.H.H Ingalls: "A type declaration and inference system for SmallTalk", *Proc. of the ACM Conf. on Principles of Programming Languages 1982.*

[CarWeg 86] L. Cardelli, P. Wegner: "On Understanding Types, Data Abstraction, and Polymorphism", *ACM Computing Surveys*, Vol.17, No.4, pp.471-522, December 1986.

[CleDDK 86] D. Clément, J. Despeyroux, T. Despeyroux, G. Kahn: "A Simple Applicative Language: Mini-ML", *Proc. of the ACM Symp. on Lisp and Functional Programming 1986*,13-27.

[CooHC 90] W. Cook, W. Hill, P. Canning: "Inheritance is not Subtyping" to appear in *Proc. of the ACM Conf. on Principles of Programming Languages 1990.*

[DamMil 82] L. Damas, R. Milner: "Principal type-schemes for functional programs", *Proc. of the ACM Conf. on Principles of Programming Languages 1982*

[DeMGab 87] L.G. DeMichiel, R.P. Gabriel: "CLOS Overview", *Proceedings of ECOOP 87.*

[HudWad 88] P. Hudak, P. Wadler et al.: "Report on the Functional Programming Language Haskell, Draft proposed standard", Research report, Yale University, Dec. 88.

[Huet 76] G. Huet: "Résolution d'équations dans les langages d'ordre $1, 2, \ldots, \omega$", Thèse de doctorat d'état, Université Paris 7, 1976.

[Johnson 86] R.E. Johnson: "Type-Checking SmallTalk" in OOPSLA '86 Proceeding *ACM SIGPLAN* Vol 21., No. 11, pp.315-321, November 86.

[Kaes 88] S. Kaes: " Parametric Overloading in Polymorphic Programming Languages", *Proc. of the 2nd European Symp. on Programming 88* LNCS 300,Springer-Verlag.

[Lang 86] B. Lang "The Virtual Tree Processor" *Esprit Project GIPE, Third review report*,September 1986.

[Liskov 81] B.H. Liskov: "Clu Reference Manual", *Lecture Notes in Computer Science* 114,Springer-Verlag 1981.

[MeyNM 87] B. Meyer, J-M. Nerson, M. Matsuo: "Eiffel: Object-oriented design for software engineering", *Proc. of the 1st European Software Engineering Conference* Sept.87, AFCET.

[Milner 78] R. Milner: "A Theory of Type Polymorphism in Programming", *J. Comput. Syst. Sci* 17 (1978), pp.348-375.

[Lunau 89] C. Pii Lunau: "Separation of Hierarchies in Duo-Talk", *Journal of Object-Oriented Programming Jul/Aug 1989.*

[RajLev 89] R.K. Raj, H.M. Levy : "A Compositional Model for Software Reuse", *Conf. Proceedings of ECOOP '89.*

[Rémy 89] D. Rémy: "Typechecking records and variants in a natural extension of ML", *Proc. of the ACM Symp. on Principles of Programming Languages 1989*

[SchaCoo 86] C. Schaffert, T. Cooper et al. : "An Introduction to Trellis/Owl", *Conf. Proceedings of OOPSLA '86*

[Suzuki 81] N. Suzuki: "Inferring Types in SmallTalk", *Proc. of the ACM Symp. on Principles of Programming Languages 1981.*

[Tofte 87] M. Tofte: "Operational Semantics and Polymorphic Type Inference", *Ph.D. Thesis* University of Edinburgh, 1987.

[WadBlo 89] P. Wadler, S. Blott: "How to make ad-hoc polymorphism less ad-hoc", *Proc. of the ACM Symp. on Principles of Programming Languages 1989*

Figure 4: Overloading schemes

$$
\begin{array}{rcl}
0,1,\ldots & \leadsto & (\omega, \{\ num : \omega\ \}) \\
"a",\ldots & \leadsto & (\omega, \{\ str : \omega\ \}) \\
= & \leadsto & (\omega \rightarrow \omega \rightarrow Bool), \{\}) \\
+ & \leadsto & (\omega \rightarrow \omega \rightarrow \omega, \{\}) \\
hd & \leadsto & (\omega \rightarrow \alpha_g, \{\ elem\ :\ \alpha_g\ \}) \\
tl & \leadsto & (\omega \rightarrow \omega, \{\}) \\
cons & \leadsto & (\alpha_g \rightarrow \omega \rightarrow \omega, \{\ elem\ :\ \alpha_g\ \}) \\
nil & \leadsto & (\omega, \{\}) \\
null? & \leadsto & (\omega \rightarrow Bool, \{\})
\end{array}
$$

A Examples

The examples given below are samples of what a prototype of the system (with recursion and a toplevel) produces. As announced, the readability of inferred types is not good. Θ-types looks like $\omega = \{p_1 : \sigma_1, p_2 : \sigma_2 \ldots p_n : \sigma_n; \phi\}$ for ADTs. Field names (symbols in L) are here explicitly given. ∇ fields are omitted. Field variables are grouped in a unique extension variable, noted ϕ. Generic variables are prefixed with the quote character(e.g. $'a$). In the following, they are written, for example:

```
SA={+:SA->SA->SA ; 'u}
```

Implementations are denoted by their unique name. The environment used in the following examples contains

- a basic type for booleans (*Bool*)
- a built-in implementation of integers, with primitives: addition (+), equality (=), and property *num*. All integer constants are overloaded, and we use the property *num* for this purpose.
- a built-in implementation of strings, with primitives: concatenation (+), equality (=), and property *str* to overload all string tokens.
- a built-in implementation of lists, with usual primitives: *hd*, *tl*, *cons*, *nil*, *null?*

The formal definition of this environment is given in Figure 4 and Figure 5.

Figure 5: Implementations descriptions

$$Integer \sim \left\{ \begin{array}{lcl} = & : & \omega \rightarrow \omega \rightarrow Bool \\ + & : & \omega \rightarrow \omega \rightarrow \omega \\ num & : & \omega \end{array} \right\}$$

$$String \sim \left\{ \begin{array}{lcl} = & : & \omega \rightarrow \omega \rightarrow Bool \\ + & : & \omega \rightarrow \omega \rightarrow \omega \\ str & : & \omega \end{array} \right\}$$

$$List \sim \left\{ \begin{array}{lcl} hd & : & \omega \rightarrow \alpha_g \\ tl & : & \omega \rightarrow \omega \\ cons & : & \alpha_g \rightarrow \omega \rightarrow \omega \\ nil & : & \omega \\ null? & : & \omega \rightarrow Bool \\ elem & : & \alpha_g \end{array} \right\}$$

Each example is chosen to exhibit a feature or a property of the type system.

Example 1 : Simple overloading

```
#let double = fun x -> x + x ;;
(* SR is happy *)
double : SA -> SA
with
SA={+:SA->SA->SA ; 'u}
```

Double may be used on any data type providing a + operation, so here

```
#(double 3, double "foo") ;;
(* SR finds Integer, String *)
(6,"foofoo") : (Integer & String)
```

Note that lexical tokens 3 and "foo" are overloaded though here only one implementation is provided for each. Static resolution had to find an implementation for the ADTs $\omega = \{num : \omega; + : \omega \rightarrow \omega \rightarrow \omega; \phi_1\}$ and $\omega = \{str : \omega; + : \omega \rightarrow \omega \rightarrow \omega; \phi_2\}$.

Example 2 : Static resolution at work

```
#fun x -> hd (cons x nil) ;;
(* SR finds List *)
fun : 'a -> 'a
```

In this example, the overloaded symbols $hd, cons, nil$ have been statically resolved since the inference determined that their resolution do not depend on the type of x.

Example 3 : Constructor/Destructor dissociation

```
#letrec map =
# fun f ->
#    (fun l ->
#        if (null? l)
#        then nil
#        else (cons (f (hd l))
#                       (map f (tl l)))) ;;
(* SR is happy *)
map : ('a -> 'b) -> SA -> SB
with
SA={hd:SA->'a , tl:SA->SA, elem:'a,
    null?:SA->Bool ; 'u}
SB={nil:SB, cons:'b->SB->SB, elem:'b, 'v}
```

The fact that the list primitives are dissociated in our environment explains this typing, where constructors and destructors of the list have been separated in the two types. A remarkable consequense is that the result list may not have the same implementation as the input list.

Example 4 : More on implementation independance

```
#letrec append =
# fun l1 ->
#    (fun l2 ->
#        if (null? l1)
#        then l2
#        else (cons (hd l1)
#                       (append (tl l1) l2))) ;;
(* SR is happy *)
append : SA->SB->SB
with
SA={hd:SA->'a, tl:SA->SA, null:SA->Bool,
    elem: 'a}
SB={cons:'a->SB->SB, elem:'a}
```

Although the input lists must contain elements of the same type, their implemention might be different. However, by construction of the program, the result list has the same implementation as the second input list.