Attaching Second-Order Types to Methods in an Object-Oriented Language

Yves Caseau
Bellcore, 445 South Street
Morristown, NJ 07960, USA
(201) 829 44 71
caseau@bellcore.com

Laurent Perron
Ecole Normale Supérieure
45 rue d'Ulm
75005 Paris, France
perron@clipper.ens.fr

Abstract

This paper proposes an extension of the notion of method as it is currently used in most object-oriented languages. We define polymethods as methods that we can attach directly to types, as opposed to classes and that we can describe with a second-order type. Two benefits result from this extension; first, the expressive power of the language is improved with better modeling abilities. Next, second-order types yield a more powerful (precise) type inference, which extends the range of static type checking in a truly extensible object-oriented language. We first show that extensible object-oriented languages present many difficulties for static type-checking and that second-order types are necessary to get stronger type-checking. We illustrate how to combine polymethods through *type* inheritance and propose a technique based on abstract interpretation to derive a second-order type for new polymethods.

1. Introduction

Object-oriented languages have two interesting features that distinguish them from other programming languages: they are *order-sorted* and *extensible*. By order-sorted we mean that their data structures (objects) are organized into a (class) hierarchy, with implicit inclusion polymorphism (inheritance). By extensible we mean that new classes can be added at any time, including dynamically with some languages. Some languages (Smalltalk [GR83], LAURE[Ca91]) are more extensible than others (C++[Str86]), since new methods can also be added dynamically. Another interesting feature of some of those languages is the *reflection* [MN88] of the organization, when classes (Smalltalk) or types (LAURE) are first-class objects. This enables (Section 2.3) the definition of light-weighted parametric polymorphic objects, using classes or types as object attributes. Generally speaking, these languages are very convenient for prototyping and yield a substantial reduction in the prototype development time.

However, the same languages often suffer from a lack of static type-checking necessary to achieve efficiency through compilation. As a consequence, these languages are rarely used for large software. Much work has been done to palliate this situation, starting at the two ends of the problem. In one case, features of object-oriented programming have been added to safe, strongly typed languages (e.g., QUEST [Car89]). In the second case, new type systems and their type inference procedures have been developed for object-oriented languages (e.g., Typed Smalltalk [GJ90]). Still, many problems remain when we want to apply static type-checking to a truly extensible, order-sorted language (e.g., the long discussion in the newsgroup *comp.object*, initiated by C. Chambers, about the *min* function, also in Section 2.1). It must be emphasized that our goal is not to identify a new language that can be statically type-checked, but to find a way to do as much static type inference/checking as possible in a knowledge representation language that already exists (knowledge representation *demands* flexibility and extensibility).

Here, we describe the solution that we have implemented in the LAURE language, based on the type system **T** described in [CP91] and the notion of abstract interpretation over a function space. If one looks at type inference as an abstract interpretation of the semantics, poor type inferences are due to an oversimplification in the image lattice when an ambiguous situation occurs. Our idea is to delay this reduction and to carry an abstract expression (a type function) instead, which will be applied to the right context, thus yielding a better decision. We have extended the usual notion of a *method* with three features (resulting in *polymethods*). First, a second-order type (a function on types from **T**) is attached to each method. Second, methods are inserted directly in the type hierarchy, instead of using the class hierarchy, and some additional relations among method parameters can be specified.

The paper is organized as follows. Section 2 deals with our motivations for introducing polymethods. We present a set of examples that are hard to type using current techniques proposed for object-oriented languages. Section 3 recalls the necessary properties of our underlying type system, developed in [CP91]. We define polymethods and present a language for second-order types. Section 4 gives a practical description of polymethods and shows how they can be used. We also give an informal description of the technique used for second-order type inference. Section 5 first gives an algorithm for inheritance conflicts and then proposes an inference procedure to compute the second-order type of each new method. The last section illustrates how polymethods solve some of the problems presented earlier.

2. Motivations

2.1 Strong Type-Checking

We consider a set of classes *integer, number, string* .... with an ordering feature ≤. Since we need to make a distinction between the property ≤ and its definition on the various classes ≤@integer, ≤@string ... we shall call the first a *feature* (following the OSF notations [Smo89] [AP90]) and the others *methods* (as in any object-oriented language [Mit90]). We suppose two important hypotheses :

- ≤ supports inclusion polymorphism (e.g., *integer* ⊂ *number*),
- ≤ is extensible (new classes *c* can be added with a method ≤@*c*).

A correctly typed definition of the *min* feature has already been given when *one* of the hypotheses holds (e.g., ABC[GMP90], Eiffel[Me89]). If we consider the following method definition, which represents the *min* feature (*entity* is the set of everything).

Example:[1] [**define** min(x:entity, y:entity) **method** -> object => [**if** (x ≤ y) x **else** y]]

[1] All examples are given using LAURE syntax. We hope that the simplicity of the examples will make the syntax self-evident. Keywords are in boldface, -> means range, => precedes the definition of a method. Messages are written using a functional syntax.

This method will not yield static type-checking, since it is defined on the class *object* (the top of the hierarchy). If it is applied to an object that doesn't have the feature ≤, a dynamic error will occur that was not detected by the compiler. If we have a truly extensible language, we can do better, we can create a class *ordered_set,* and define a method ≤@ordered_set to be sure that every member of an ordered set has the feature ≤.

```
Example:   [define ordered_set class subsets (number string),
                with method( ≤(x:ordered_set) -> boolean
                                         => [error "≤ is not defined on ~S and ~S", self, x])]
```

Assuming this is allowed by the language (which is rare), this is still not enough, since we will not catch an error as simple as *min(1,"john").* Because of the inclusion polymorphism, the solution proposed for the Eiffel language, which is to declare the two arguments *self* and *x* of *min* as having the *same* type, doesn't work here : min(1,3.5) is a valid instruction. Similarly, we cannot represent the type of *min* with a disjoint union like

$$min : (integer \times integer \rightarrow integer) \cup (number \times number \rightarrow number) \cup \ldots$$

because this would miss the link between *min* and ≤, any new class can potentially use the *min* feature. A complex solution would be to store an explicit disjunction and a graph of dependencies, which would be used to re-compute the methods' type dynamically. We have found the use of second-order types to be a much more elegant way to represent the intimate relationship between the two methods ≤ and *min* in the type system.

2.2 Order-Sorted and Heterogeneous Sets

The question of heterogeneous structures, such as sets, yields many interesting questions when we try to develop static type-checking inference for a knowledge representation object-oriented language, because of the inclusion polymorphism [CW85]. Here, we consider the case of lists as an example; roughly speaking, there are two ways of grasping the problem.

- Typed lists: each list $(a_1 \ldots a_n):t$ has a given type t such that all members a_i of the list are of type t. This is the solution of ML [HMM86], or Machiavelli [OBB89], because it is easier to get type safety on list operations this way. As a consequence, there are many empty lists *():integer, ():string* ... that are all different.
- Untyped lists: a list has no given type, except the generic type *list*. If we have a type lattice, we can always associate the type $\bigvee_{i=1}^{i=n} t_i$ to the list $(a_1 \ldots a_n)$, where t_i is the type of each member a_i. As a consequence, there is a unique empty list *()*. This is a simpler solution, and it gives a simpler semantics to set operations (as opposed to the special meaning of set union in Machiavelli [OBB89]).

Although the first solution is simpler, it is not relevant for knowledge representation object-oriented languages, as is illustrated by the following example. Suppose that we have two classes, *person* and *employee*, and that John and Mary are employees and Peter is just a person (*employee* is a subclass of *person*). What happens if we delete Peter in the list (John Mary Peter)*:person*? If we define delete with the following parametric signature.

$$delete: \quad list[t] \rightarrow list[t],$$

delete(Peter,*add*(Peter,(John Mary)) is different from (John Mary), because they have different types. If we give a finer description such as

$delete\colon\ list[t] \rightarrow list[t']\, with\ t \prec t'$,

then there is no reason why *delete*(John,(John)) would be of type list[*person*]; we are back to the second solution, where there is a unique empty list and where lists have no fixed types. Therefore, we chose the second solution for an object-oriented solution because it is more natural. However, this will make static type-checking more difficult since list[*integer*] and list[*string*] are no longer disjoint, their intersection is $\{()\}$. Our solution is to attach methods directly to types, so that the case of the empty list can be singled out, and to use second-order types to capture parametric type inference.

2.3 Optimization and Parametric Polymorphism

Let us now consider the stack example, as can be written with a reflective object-oriented language such as LAURE. We attach a slot *of* to each stack object, which represents a type for all objects stored in the stack (the rest of the representation is as usual: an *index* and an array *content*).

```
[define stack class
    with    slot( of -> type, default ∅),
            slot( index  -> integer, default 0),
            list_slot(content),
            method( top => last(content(self))),
            method( push(x:entity)
                => [if (x ∈ of(self)) [do  (content(self) put_at  index(self) x]
                                           (index(self) is (index(self) + 1))]
                    else range_error( arg  x, set  of(self))]), ...
[define my_stack  stack of integer]
```

This example shows how a simple polymorphic stack can be implemented with such a language, which entirely relies on dynamic typing. Here, types are seen as sets (hence the test *(x ∈ of(self))* for checking if *x* is of type *of(oself)*) and a type error is created when the test fails. More details about the power of exceptions as first-class objects can be found in [Do90].

However, this leaves us with two problems. First, we want to introduce parametric type inference [CW85], such as in [OB89], so that we can deduce that *top(my_stack)* is an integer. In addition, we would like to perform static type checking to improve safety, and we would like to generate better code (compiler optimization) when the argument types are found to be correct. This implies that the test*(x ∈ of(self))* must be taken out of the code and placed in the method's type definition. We prefer to extend the method notion as opposed to extending the object model with class parameters [CP91].

3. Background: The LAURE Type System

3.1 A Set-Based Type System

In this section we recall some of the interesting properties of the type system presented in [CP91], which are necessary to build polymethods. We see objects as nodes in a *semantic network*, without any structure (such a network, which represents the state of the object system, is denoted by a function ν of the set S).[2] Objects are organized into a class lattice[3] $(C,<)$, which is *reified* [FW84] (the set of classes C is included into the set of objects O). Each class c represents a set of objects, written $\chi_\nu(c)$. Classes are used as types, but we also add the following type constructions: finite enumerations $(\{o_1,\ldots,o_i\})$ allow the compiler to perform optimizations based on recognition of constants; disjunctive types permit type inference without loss of information; powerset types (*set_of*) are introduced to optimize set operations and parametric types ($p{:}e$, where p is a parameter from a set P, is the set of objects whose parameter p belongs to e). We define types through a type expression language E, where each expression represents a set.

<type:E> ::	$\{o_1,\ldots,o_i\}$ | C | **set_of**(<E>) |
	<E> $\cup$ <E> | <E> $\cap$ <E> | <E> - $\{o_1,\ldots,o_i\}$ | <P>:<E>

Example: • *{string}* $\cup$ *integer}* represents the sets of strings or integers,
• *{list - {()}}* represents the set of non-empty lists.

Similarly, we write $\chi_\nu(e)$ the set of objects represented by the type expression e in the object system $\nu \in S$. In [CP91] we define the notion of a licit evolution of a system, which gives a precise meaning to the term *extensibility*. We have shown that we can define a type *normal form* from N, which supports two syntactical operations $\prec$ and $\wedge$, that represent, respectively, subtyping and type combination. We define our type system **T** to be the quotient of N by the equivalence relation $\approx$ $\left(t_1 \approx t_2 \Leftrightarrow (t_1 \prec t_2)\&(t_2 \prec t_1)\right)$ and we get the following result.

Theorem: *$(T, \prec, \wedge)$ is a complete lattice and subtyping is inclusion:*
$\forall t_1, t_2 \in T,\ t_1 \prec t_2 \Leftrightarrow$ for every licit system ν, $\chi_\nu(t_1) \subset \chi_\nu(t_2)$

In addition [CP91], we have defined two interesting functions on types: ε and ρ. Informally, ε is a meta-type extraction procedure ($\varepsilon(t)$ is a valid type for all members of a set of type t) and ρ is a parameter-extraction procedure ($\rho(p,t)$ is a valid type for the object y as soon as $y = \nu(p)(x)$ and x is an object of type t). These functions are necessary to build the lattice structure, but also play an important role for type inference.

3.2 Features and Polymethods

Objects are described with *features*, which represent relationships among objects and operations that can be performed on those objects. The set of features F has a distinguished subset A of *attributes*, which are binary relations among objects stored in the object system ν. A itself has a distinguished subset P that represents a set of *parameters* (introduced previously), which associates one unique value to each object.

We can now introduce *polymethods*, used to define features from F.

[2] The set S contains functions that associates binary relations on O to each attribute [CP91].

[3] We assume that the taxonomy $(C,<)$ is a lattice, without any loss of generality since we can efficiently extend any hierarchy into a complete lattice. Notice that this does not preclude classes from being mutually exclusive [CP91].

Definition: *A polymethod is a tuple $m = (r(m) \in F, \sigma(m), f(m), \kappa(m))$:*

- *$r(m)$ is the feature that m is defining,*
- *$\sigma(m) = t_1 \times ... \times t_n$, and is called the* ***signature*** *of m, where $t_i \in T$.*
- *$f(m)$ is a function (the* ***definition****) from $S \times O^{n-1} \rightarrow S \times O$ such that:*
  $\forall v \in S, \forall a_1 ... a_{n-1} \in O,$
  $\forall i, (a_i \in \chi_v(t_i))\ \&\ f(m)(a_1 ... a_{n-1}) = (v', y))$
  $\Rightarrow v'$ *is a licit system* $\&\ y \in \chi_{v'}(t_n)$.
- *$\kappa(m)$ is a* ***second-order*** *type for m:*
  $\forall v \in S, \forall x_1, ..., x_{n-1} \in T, \forall a_1, ..., a_n \in O,$
  $(\forall i, (x_i \prec t_i)\ \&\ (a_i \in \chi_v(x_i))) \wedge f(m)(a_1, ..., a_{n-1}) = (v', y)) \Rightarrow$
  $y \in \chi_{v'}(\kappa(m)(x_1, ..., x_{n-1})$

The signature tells when the polymethod can be used to compute a given feature. Each polymethod is a restriction of the feature $r(m)$ ($t_1 \times ... \times t_{n-1}$ is called the domain of the restriction, t_n is called its range) and is responsible to know when it is valid.[4] This definition is an extension over the usual notion of a method in two ways: we attach polymethods to types as opposed to classes only and we complete the polymethod definition by a second-order type, which supports better type inference.

Example: • Identity is a feature defined with one polymethod (Id, σ, f, κ):
f = (λx.x),
σ= Object $\times$ Object,
κ = (λx.x).

• Fibonacci is a feature defined with two polymethods
(fib, {0,1} $\times$ Integer, (λx.1), (λt.{1}))
(fib, Integer $\times$ Integer, (χ.(fib(x-1) + fib(x - 2))), (λt.Integer))

Definition: *A feature f is a collection of polymethods m with same arity and $r(m) = f$. A feature is well-defined if for any tuple of object $(x_1, ..., x_n)$ there exists a unique smaller restriction to apply (restrictions are naturally ordered by the inclusion product order on their domains).*
In this case, f represents a function from $S \times O^{n-1} \rightarrow S \times O$ obtained by applying to each object tuple the unique smallest polymethod that applies.

The underlying assumption is that a restriction on a smaller domain overrides a more general restriction.[5] Transforming any collection of polymethods into a well-defined feature is the goal of multiple inheritance conflict resolution, as described in Section 5.1.

3.3 Second-Order Types

Second-order types are type functions (usually lambda-abstractions) attached to polymethods to represent the relations among the types between the input parameters and the result of the polymethod. A second-order type is a finer type description than a signature. This means if we apply $\kappa(m)$ to a type tuple $(x_1,...,x_{n-1})$ that contains an

[4] As opposed to each class knowing which feature can be used; this separation between organization and description makes organizing simpler.

[5] For instance, if we say $f(x) = x + 1 \wedge f(0) = 0$, the second equation is used to define f(0) rather than the first.

object tuple $(a_1, ..., a_{n-1})$, we will get a type that must contain the result of applying m to the object tuple. Fore instance, we can always use the canonical second-order type $(\lambda x_1 \ldots x_{n-1}.range(m))$ obtained from the signature. Here are some other valid second-order types.

Examples:

$\lambda X.X$	which is a second-order type of the *identity* polymethod,
$\lambda X.\ \varepsilon(X)$	which is the second-order type of the *car* polymethod (first member of a list),
$\lambda X.\ \rho(of,X)$	which is the second-order type of the *top* polymethod.

Our first motivation to introduce second-order types is to associate them with primitive polymethods to get better type inference. For instance, with the previous information, the compiler will know that the type of Id(e) is the same as the type of the instruction e.

However, we also want to infer such second-order types for new polymethods. For this purpose, we define a language of functional types $\Lambda(\mathbf{T})$ as lambda-abstractions over the set of types $\mathbf{T}$, using the operations introduced in the previous sections and a type inference function ψ. This function is defined in [CP91] so that $\psi(f, t_1, ..., t_n)$ is a type that will contain the type of the result of applying the feature f to objects of types $t_1, ..., t_n$. We will give an informal description of ψ in the next section.

```
Λ :: λx1 ... xn . <exp(x1, ..., xn)>.

<exp(x1, ..., xn)> :: xi | T | if( <test(x1, ..., xn)>, <exp(x1, ..., xn)>, <exp(x1, ..., xn)>) |
      ψ( <F>, <exp(x1, ..., xn)>, ... , <exp(x1, ..., xn)>) |       feature inference
      ε(<exp(x1, ..., xn)>) |                                        powertype inference
      ρ(<P>, <exp(x1, ..., xn)) |                                    parametric inference
      <exp(x1, ..., xn)> ∧ <exp(x1, ..., xn)> |                      lattice join
      <exp(x1, ..., xn)> ∨ <exp(x1, ..., xn)>                        lattice meet

<test(x1, ..., xn)> :: <exp(x1, ..., xn)> = <exp(x1, ..., xn)> |     type equality
      <exp(x1, ..., xn)> ≺ <exp(x1, ..., xn)>.                       type subsumption
```

Each lambda-abstraction of Λ with n variables represents a function from $\mathbf{T}^n \rightarrow \mathbf{T}$. Such a function is a second-order type for a polymethod m, if it can be used to predict the type of the result of applying m. to a set of arguments with known types.

4. Informal Description of Polymethods

In this section we explain why polymethods are a useful extension from methods. We also give an informal description of the inference strategy to derive second-order types for new methods. A more precise description will follow in Section 5.

4.1 Attaching Methods to Types

The first original feature of polymethods is to be attached to a type as opposed to a class. Since classes are types, this is a generalization of method attachment, which is made possible because our type system has the same inclusion lattice structure as the class lattice. The practical advantage is that we can attach the method more precisely where it is needed, which yields shorter code (in a traditional method we would have to add some code to test that the first argument satisfies some extra condition) and better readability. Here are some examples that show how we can now use the type hierarchy to define methods.

- *Attaching a method to a union.* With a traditional OO language, sharing code in only performed through inheritance. If we have two separate classes A and B that could share a method, we need either to duplicate the code, or create an additional class C from which A and B will inherit and to which we attach the method. This situation has two major drawbacks; first, it is not always possible to insert the new class C. Let us consider a toy example where we want to define a new method *very_long?* which applies both to strings and lists. Since the two classes *string* and *list* have been created previously, neither C++ or Smalltalk will allow the definition of a new class from which they would inherit (in a simple manner). The second drawback is that code sharing yields the definition of multiple classes, which confuse the initial design of the taxonomy. Because the taxonomy is the backbone of software organization in an OO language, this sort of "overloading" should be avoided. Using a polymethod is a simpler solution,

  ```
  [define very_long(x:{string + list}) polymethod -> boolean, => length(x) > 80],
  ```

  which allows sharing without useless complexity.

- *Attaching a method to a single object or a given collection of objects.* Another problem that we have encountered frequently while writing LAURE applications is methods that apply to a single object. This occurs either because the object is one-of-a-kind (e.g., a keyword), or because it needs a special behavior. The usual solution is to create a special class in the first case and to insert a conditional test in the generic method when the second case occurs. We already mentioned the drawbacks associated with the creation of a new class that is not necessary from a design point of view. Similarly, treating exceptional situations with conditional statements is not an optimal solution and becomes confusing if there are many such "special" objects. Since finite collections are types in LAURE, we may attach a polymethod to one object or to a collection of objects (*a la* MODULA2) as in

  ```
  [define fact(x:{0}) polymethod -> integer, => 1]
  [define fib(x:{0,1}) polymethod -> integer, => 1].
  ```

- *Handling exceptional objects.* The previous example showed how exceptional objects could be handled positively (by saying how to deal with them). Type difference can be used to deal with them negatively. For instance, we can define inversion on non-zero numbers.

  ```
  [define inverse(x:(float - {0.0})) polymethod -> float, => (1 / x)].
  ```

  Dealing with the exception (0) in such a way is safer than relying on a specialized method defined on {0} and is more elegant than using a conditional statement.

- *Attaching a method to a parameterized type.* Last but not least, we may now attach methods to complex type expressions that represent subsets of a generic class. For instance, the sum feature should be defined only on lists that contains numbers (integer, float ...). This is captured easily with a polymethod

  ```
  [define sum(l:(list & {number set_of})) polymethod -> number,
      => [let n as 0, [for x in l, n <- (n + x)], n]]
  ```

  The same example would hold with {stack & {of in {number}}}.

Such an improvement in expressive power is bound to entail additional complexity. The two main issues are the resolution of multiple inheritance conflicts and the efficient compilation of polymethods. The efficient evaluation is actually a sub-problem of the compilation, which we consider less important since our focus with LAURE is to bring flexibility in the interpreted mode but deliver performance in the compiled mode (actually LAURE's interpreter is faster than equivalent COMMONLISP interpreters).

Inheritance conflicts are more frequent with polymethods since the type lattice is more complex and there is a greater chance for two polymethods to have signatures with a non-empty intersection. However, we can re-use the strategy that was developed for LAURE previously [Ca89], where resolution is done by applying a commutative and associative combination operator over the lattice structure. We used to rely on the class lattice structure of LAURE, but the solution still works with a more complex lattice structure (cf. Section 5.1).

We deal with the compilation issue in the same way, using the techniques that were developed to compile methods. When LAURE tries to compile a message using the feature *f*, it uses the type inference module to determine a type for each sub-expression. If the feature *f* is closed [Ca89] (which really means that we allow static binding), LAURE will search if the types of the arguments yield a unique method *m* for the feature *f*. This is the case when

(1) the argument signature is included in the method *m*'s signature,

(2) the argument signature has an empty intersection with the signature of any other method that is not bigger than *m* according to order defined in Section 3.2 (*f* is well-defined).

Practically, this is often the case, which explains why compilers such as Typed Smalltalk [GJ90] or LAURE can improve performance up to C++ level. However, when this does not occur, we must rely on dynamic binding and determine at run-time which method to call. The only way to get rid of this last case is to build a statically-typed language, but this creates a large set of problems as we have seen in the first section, and does not seem relevant for a prototyping/ knowledge representation language. By combining a powerful type inference mechanism (such as the one we are describing) with the flexibility of dynamic typing/binding (when required), we can combine advantages and win on both aspects.

4.2 Attaching Type Expression to Methods

The second original feature of polymethods is the ability to receive a second-order type. In the rest of the paper, we will concentrate on how such a type can be derived automatically. Here we would like to show that second-order types can be used directly by an (advanced) user to help the compiler. When defining a polymethod, the user writes an expression introduced by the keyword =>, with an implicit lambda-abstraction; it is also possible to write another expression, introduced by the keyword :=>, which represents the type of the result using the method variables to represent the types of the parameters. Let us consider an example where we define addition over lists, sets and strings. We also want the compiler to detect errors (one cannot add a string and a set) and to make polymorphic type inference (adding two strings will produce a string). Thus we write the following definition:

```
[define plus(x:{set + {list + string}} y:{set + {list + string}}) polymethod -> {set + {list + string}},
    => [case x   set  (x + y),
                 list (copy(x) append y),
                 string (x append y)],
    :=> [if (x ≤ set) [if (y ≤ set) set else {}]
         else_if (x ≤ list) [if (y ≤ list) list else {}]
else_if (x ≤ string) [if (y ≤ string) string else {}]
else entity]]
```

In the second expression (after :=>) the variable x and y represent the types of the two arguments. Here we only use conditional expression and the type lattice order ($\prec$), but we could build more complex type expressions using the operators *in*: and @ (in(x) is the LAURE equivalent of $\varepsilon(x)$ and (x @ p) the equivalent of $\rho(x,p)$). We can use LAURE to write second-order types because types are first-class objects in LAURE.[6]

Indeed, the goal of second-order type inference is to produce the type function automatically for such an example. However, there are still cases when the user must give the second-order type, because it could not be deduced. This is, for instance, the case for primitive methods or for methods that use heterogeneous data structures, such as in the two examples

```
[define car(l:list) method -> entity, => #'car_list , :=> in(l)]
[define top(x:stack) method -> entity, => last(content(x)), :=> (x @ of)]
```

Second-order types are used at compile-time during type inference. If we remember from the last section how messages were compiled, the second-order type will be used when the compiler has found which method to apply. If such a method m has a second-order type $\kappa(m)$, the compiler will apply this function to the tuple of types of the arguments in the message to compute the type inferred for the result of the message, as opposed to simply using the range of the method. Thus, if we compile the message plus("aa","bb"), the compiler will infer the type *string* and not simply {set + {list + string}}. The combined process of finding which method of the feature f to apply for a set of arguments $(x_1, x_2, \ldots x_n)$ with given types $(t_1, t_2, \ldots t_n)$ and of computing the predicted type of the result is embodied in the function ψ (i.e., $\psi(f,t_1, t_2, \ldots t_n)$ is the type inferred for the expression $f(x_1, x_2, \ldots x_n)$.

4.3 Second-Order Type Inference

In this section we would like to describe informally the principles used to build automatically second-order types for methods. A more precise description will follow in the next sections. A type inference procedure is an abstraction mechanism, that can be seen formally as an abstract interpretation. This means that we abstract values into set of values and we predict the result of an expression with a set that will contain the actual result. For instance, a usual abstraction of a method is to check if the types of the arguments are correct and return the range of the method as the type inferred for the result.

Precision in type inference is lost when the abstraction is performed at a coarse level. For instance, if the identity method (λx.x) is abstracted into *Entity* $\rightarrow$ *Entity*, then the type inferred for identity(1) is *Entity* (as opposed to *integer* or {1}). To get better type inference, we need a better abstract domain that can carry more information. We

[6]This idea that the user would like to help the compiler with a second-order type has actually been extended in the LAURE language to a further degree. When the user gives the type function, this function may either return a type, in which case it is understood as a second-order type, but it can also return a method, in which case the function is seen as an optimization rule. The method returned by the type expression is substituted to the original method in the compilation process. This allows to perform optimizations based on type information in a way that is (somehow) elegant and extensible (the user can add his own rules).

can also see that a better inference for identity(1) depends both on the abstraction for the identity method but also from the actual type of the argument.

Thus, using functions over types as an abstraction domain is a natural method to improve the precision of type inference. This is the principal intuition behind the use of second-order types. We want to abstract each method into a lambda-abstraction that gives the type of the result as a function of the type of the arguments. Since a method is defined by an expression of the language, this means that the type inferred for an expression *e* must not be a simple type (e.g., *Entity* in the case of the identity method) but rather a type expression that uses the types of the method arguments as its own arguments (e.g., *x* for the same identity method). Then we can obtain a second-order type by lambda-abstraction (e.g., $\lambda x.x$ for the identity method).

Using a functional domain allows some typing decision to be delayed until the type is actually applied, thus we can describe it as a lazy abstraction since part of the work is done later. For instance, type inference on the message identity(1) will return $\{1\}$, when the type of the identity method ($\lambda x.x$) is applied to the type of the argument 1 (which is $\{1\}$).

Practically, we use an abstraction method Ξ which translates an instruction on values into an instruction on types, using the basic type constructors and some type inference methods (such as ψ). This method is implemented on each kind of LAURE instruction (all LAURE instructions are first class objects). The result of each method is another LAURE instruction, which represents an expression from $\Lambda(\mathbb{T})$.

5. Using Polymethods

5.1 Framework for Inheritance Conflict Resolution

In order to get *well-defined* features, we need to solve "multiple inheritance conflicts". To do so, we assume the existence of a combination property, which takes two methods of the same feature and creates a third.

Proposition: *There exists a commutative and associative combination operation, written $\otimes$, which takes two methods of the same feature and returns a new method whose domain is the intersection of the domains.*

The actual combination operation that we use in the language depends on the degree of description that we have for each method. For instance, if a set value is returned by m_1 and m_2 , a natural convention is to return the union of the two sets for the method $m_1 \otimes m_2$.[7] Another common rule is to choose m_1 over m_2 if the domain of m_1 is included in m_2's domain. The more information we have on each method, the better the resolution conflict (thus, *reflection* is interesting, because we can enrich the description of a method). Last, we need to introduce an *error* value, which is the only way to deal with some situations (e.g., $\forall$ x $\in$ $\{1,2\}$ f(x) = 1 & $\forall$ x $\in$ $\{2,3\}$ f(x) = 2). We now use the combination operation to solve all virtual conflicts.

[7] This point of view comes from a knowledge representation and AI background. Conflict resolution is a semantic problem that does not have a unique well-founded solution. What is described here is a framework which allows us to build a consistent conflict resolution strategy from a few semantic rules given by the user [Ca89].

Strategy: *If f is a feature with two methods, whose domains have a non-empty intersection D such that there is no other restriction with domain D, we add a new method obtained by combining two such methods with ⊗. We repeat this process until there are no such pairs of methods. The result of the iteration is independent of the choice of a particular pair at each step. The result is a well-defined feature.*

The Church-Rosser property results from associativity and commutativity of ⊗. This strategy has many advantages. First, we get a truly declarative semantics for multiple inheritance, which is more satisfying than operational ones based on the order in which the taxonomy is explored. Because of the lattice structure, each conflict is solved once at the "higher" possible level. Moreover, the conflict resolution strategy can be enriched with new rules for method combination. An important corollary is that for each well-defined feature, the set of method's domains is closed by intersection, which is close to the condition imposed in OBJ2 [GM86].

5.2 Type Inference and Abstract Interpretation

The LAURE language is an imperative language, built around the notion of message passing, which associates a function written $[\![e]\!]$ from $S \to O \times S$ to each expression e of the language. The input parameter is the state of the system and the output is a pair (resulting value, resulting state). We have developed a type inference procedure as an *abstract interpretation* [CC77] of the semantics [AM91] [CP91]. We abstract each semantics function $[\![e]\!]$ as a type $[\![e]\!]_\Gamma{}^*$ from **T**, which represents the type expected for the result of evaluating e. The fact that each variable is typed in LAURE is represented by the context $\Gamma = \{a_i : X_i,\ i = 1\dots n\}$, where a_is are variables and X_i are types. The definition of the abstraction is given by the fact that the type lattice is a sub-lattice of O's powerset. The consistency of the abstract interpretation [CC77] guarantees that $\forall\, v \in S, [\![e]\!]_1 \in \chi([\![e]\!]_\Gamma{}^*)$. This standard abstraction is shown as *type inference* in Figure 1.

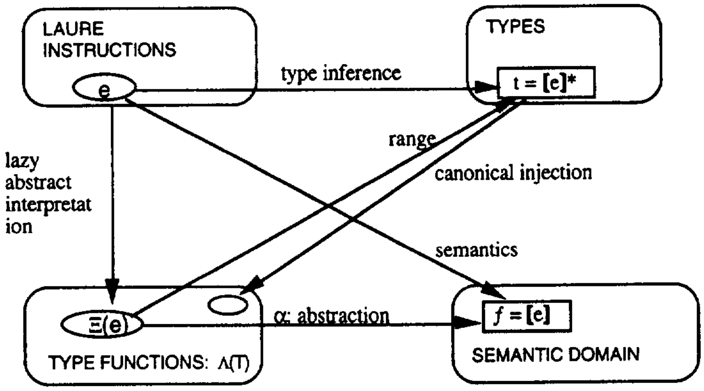

Figure 1: Two Schemes of Abstract Interpretation

However, the problems that we have presented in the first section can be characterized as an oversimplification in the abstraction process. Our abstract domain is too limited to reflect the complex type information needed for such tricky problems. This is why we have introduced second-order types, as a more complex abstract domain (the canonical second-order type presented previously can be seen as a canonical injection from the simpler abstract domain to the other, as in Figure 1). We call this a *lazy* abstraction because representing the type of an instruction as a function over types avoids making simplifications too soon and delays the decision until a finer context Γ is known. For instance, the type obtained for the identity will be (λt.t) as opposed to Object → Object. The abstraction arrows from the TYPE FUNCTION abstract domain to the semantics domain in Figure 1 reflects the fact that each second-order type is an abstraction of the semantics (cf. the definition of a second-order type in Section 3.2)

5.3 Lazy Abstract Interpretation with Λ(T)

The key step to second-order type inference is to "abstract" the abstract interpretation rules into functions from Λ(**T**). If *e* is an expression with free variables $a_1, \ldots a_n$, we consider the types $x_1, \ldots x_n$ of those variables as meta (type) variables and we look to express $[\![e]\!]^*$ as a function of $x_1, \ldots x_n$, using Λ(**T**). Therefore, we shall build a function Ξ from the set of LAURE expression to Λ(**T**) such that

$$\forall t_1,\ldots,t_n \in \mathbf{T},\ \Xi(e)(t_1,\ldots,t_n) = [e]_\Gamma^*,\ where\ \Gamma = \{a_i : t_i,\ i = 1\ldots n\}.$$

We have extended the class lattice with two additional sets: *error* and ***ambiguous*** = *error* ∨ *object* . $[\![e]\!]_\Gamma^*$ = *error* means that the execution of *e* produces an error; $[\![e]\!]_\Gamma^*$ = *ambiguous* means that the execution of e may produce an error.

Ξ is built by structural induction over the LAURE language, in a similar manner as the abstract interpretation is defined in [CP91]. We know a bound for each type variable x_i (since $x_i \prec X_i$), so we carry the context $\Gamma = \{a_i : X_i,\ i = 1\ldots n\}$. Here is a first (simplified) example for applying Ξ to a conditional statement[8].

$$\begin{aligned}\Xi_\Gamma(\ [\mathit{if}\ e_1\ \ e_2\ \mathbf{then}\ e_3\ ]) = \ & \text{if}\ [e_1]_\Gamma^* \leq \mathit{boolean} \\ & \qquad \text{then}\ (B \wedge C) \mid B = \Xi_\Gamma(e_2),\ C = \Xi_\Gamma(e_3) \\ & \text{else\_if}\ [e_1]_\Gamma^* \leq \mathit{error}\ \text{then}\ \ \mathit{error} \\ & \text{else}\ \ \mathit{if}(A \cap \mathit{boolean} = \varnothing,\ \mathit{error},\ B \wedge C) \\ & \qquad \mid A = \Xi_\Gamma(e_1),\ B = \Xi_\Gamma(e_2),\ C = \Xi_\Gamma(e_3)\end{aligned}$$

This means if the test is known to return a boolean we return the union of the types of the two branches. Otherwise, we create an *if* structure that will test the correctness of the test against the type arguments. This is a good example of laziness: when there is some ambiguity, we carry the test in the result as opposed to making a hasty decision.

We handle messages in a similar manner using the ψ operation. The semantics associated with a message $f(a_1,\ldots,a_n)$ is to apply the restriction of the feature *f*, which is relevant (*f* is well-defined) to the tuple $[\![(a_1,\ \ldots,a_n)]\!]$ (evaluated from left to right). In [CP91], we have defined a function ψ for message type inference and shown the following result

$$\forall t_1,\ \ldots,\ t_n \in \mathbf{T},\ \forall a_i \in t_i\,,\ \forall v,\ [\ f(a_1\ ,\ldots,a_n\ )]\ (v)_1 \in \psi(f,\ t_1\ ,\ \ldots,\ t_n).$$

This means we can use ψ for direct type inference

$$[\ f(a_1\ ,\ldots,a_n\ )]_\Gamma^* = \psi(f,\ [a_1]_\Gamma^*,\ [a_2]_\Gamma^*,\ \ldots,\ [a_n]_\Gamma^*).$$

Since ψ was introduced as a symbol of Λ(**T**), we can define

[8] In the rest of the paper, we write E(x,y,...) | x = a, y = b, ... for the expression E from Λ(T) obtained by substitution.

$\Xi_\Gamma(f(e_1, e_2 \dots, e_n)) = \psi(f, A_1, \dots, A_n) \mid A_i = \Xi_\Gamma(e_i)$.

Thus, instead of simply associating to the message a type that would contain its results, ***we associate an expression that we can use to compute a better type for each variable context Γ.*** The previous definition has been extended in our current system to catch finer inference (e.g., using the test in an *if* statement).

5.4 Application to New Polymethod Definitions

We can use the function Ξ for classic type inference and compiler optimization [CP91]. In addition, we can also use it to infer a second-order type for a new method defined by some variables and a LAURE expression, according to the following result

Theorem: *If $(a_1,\dots, a_n)$ are some variables, $X_1,\dots,X_n$ are some types and $e(a_1,\dots,a_n)$ an instruction made from these variables, we can define a method m from e with:*
- $f(m)(x_1,x_2,\dots,x_n,v) = [\![e(x_1,x_2,\dots,x_n)]\!](v)$,
- $\kappa(m) = \Xi_\Gamma(e)$ with $\Gamma = \{a_i : X_i,\ i = 1\dots n\}$
- $\sigma(m) = X_1 \times \dots \times X_n \times \kappa(m)(X_1,\dots,X_n)$.

The important point is that $\kappa(m)$ (= $\kappa(e)$) can be represented by a function from Λ which is "compiled" from the expression *e*. Here is a short example that shows how a second-order type is inferred

Example: • Add1 is defined by one variable *self* and one instruction (*self + 1*) :
$f(m) = (\lambda x.x+1)$,
$\sigma(m) =$ Number × Number,
$\kappa(m)(x) = [\![\text{(self+1)}]\!]_{\{\text{self:x}\}}{}^* = \psi(+, x, \{1\})$.
This second order type means that Add1 returns an integer when applied to an integer and a number when applied to a number.

To avoid problems, we use the canonical second-order type when we need it inside a recursive definition of a method. A better solution is to produce a fix-point upper bound of the second-order type, which we are currently experimenting.

From a practical perspective, $\Lambda(\mathbf{T})$ is included in the LAURE language through reflection of the type system (all type operations, such as ψ, are methods implemented on objects that represent LAURE types). This means that the result κ(m) of the type inference procedure is simply a LAURE instruction, from which a function is made by lambda-abstraction. This means that the procedure for second-order type inference, which is given a LAURE object representing the instructions (*self* + *1*), will not produce a complex term but rather another LAURE expression (as an object) which is used to produce the second-order function. Here are some example of such a transformation:

```
self ⟶ self
(self+ 1) ⟶  range_message(+,self,{1})
[if  (self ≤ x) self  else x] ⟶  [if  ((range_message(≤,self,x) glb boolean) = {})  {} else (self lub x)]
```

The transformation is implemented by structural recursion as an abstract interpretation. Because we do not deal with recursive functions yet, the complexity of type inference is linear in the size of the program and polynomial in the size of the class taxonomy.

6. Application to Common Problems

6.1 The Min Method

We can now consider the *min* feature and see how second-order types can be used to get a proper type description. We assume here that ≤ is a primitive feature, defined by methods whose second-order types are derived from their signatures. The second-order type inferred by LAURE for the *min* method will be

$$\Xi(\ [if\ \ (self \leq x)\ self\ \ \ else\ x]\ )_{\{self:entity,\ x:entity\}}.$$

Since $[\![\ (self \leq x)\ ]\!]^*_{\{\text{self:entity, x:entity}\}} = \psi\,(\leq,\text{entity},\text{entity}) =$ *ambiguous*, we get

$$\lambda(x,y).\ if(\ \psi(\leq,\ x,\ y)\ \cap boolean = \emptyset,\ error,\ x \vee y)).$$

With this second-order type, the compiler can both perform polymorphic type inference and detect errors, while the user may extend the feature ≤ by new methods (such as a lexicographic order on lists introduced in the next section). Here are some of the strong points.

- Strong type checking is possible: *min*(1,"a") or *min*("a",2.3) will be detected as errors because ψ(≤, *integer*, *string*) = *error*. This would, of course, be also true for a message *min*(*x*,*y*) where we simply know that *x* and *y* belong to two types that are not comparable.
- *Min* is a truly polymorphic function: if we find the expression *min*(*x*,*y*) in an environment where the type of x and y is very imprecise (e.g., Object), the inferred type will simply be Object. If x and y are known to be integers, the inferred resulting type will be integer.
- This is an extensible description: if ≤ is augmented by a new method, the type for *min* is still valid without any re-computation. The dependency between ≤ and *min* is represented in the second-order type.

6.2 List Operations and Polymethods

Let us now consider the case of lexicographic order on lists, as defined the following method.

```
Example:   [define (l:list ≤ x:list) method -> boolean,
               => [   if  (l = {}) true
                      else_if  (x = {})  false
                      else [or (car(l) ≤ car(x)), [and (car(l) = car(x)), (cdr(l) ≤ cdr(x))]]]]
```

The second-order type inferred by the system will not detect that the expression (*x* ≤ *y*), where *x* is of type {*list* ∧ *set_of(integer)*} and *y* is of type {*list* ∧ *set_of(string)*}, is a type error. Actually it is not ! *x* and *y* could be two empty lists (cf. Section 2.2) and the message (*x* ≤ *y*) is legitimate. This means that we lose some safety in type-checking when we take a "truly object-oriented, extensible" point of view on lists. On the other hand, [MMM91] shows that if the good type-checking properties of ML are kept, some aspects of object-orientation must be simplified. However, here we can get a better typing if we use polymethods to distinguish the case of the empty list.

Let us consider the following definition.

```
Example:  [define (x:{()} ≤ y:list) polymethod -> boolean, => true]
          [define (x:{list - {()}} ≤ y:{()}) polymethod -> boolean, => false][9]
          [define (x:list ≤ y:list) polymethod -> boolean,
           => [or (car(x) ≤ car(y)), [and (car(x) = car(y)),(cdr(x) ≤ cdr(y))]]]
```

The second-order type inferred for the last method is

$$\lambda(x,y).if(\ \psi(\leq,\varepsilon(x),\ \varepsilon(y)) \cap boolean = \emptyset,\ error,\ boolean)$$

This allows the system to detect typing errors such as

```
((1 2 3) ≤ ("aa")) ; (((1 2) (2 3)) ≤ (((1)))) ;
```

Obviously, this only happens when the compiler can find out if the list is empty or not. This is done by extending the type inference rules and by defining primitive methods, such as cons or append, with more detailed signature.

6.3 Stacks

The last feature of polymethods is the ability to add preconditions on methods' arguments that bind the type of one argument to another. The preconditions that can be added are of the form $(x \in p(y))$, where x and y are two methods arguments and p is a parameter. Syntactically, the special form p(y) is used in place of a type in the definition of the typed variable x. For instance, here is the revised stack example, using polymethods

```
[define of parameter]
[define stack class with slot(of -> type, default ∅) ...]
[define top(s:stack) polymethod -> integer, => last(content(s)), :=> (s @ of)]
[define push(self:stack, x:of(self)) polymethod -> void,
     => (content(self) put_at index(self) x),
        (index(self) is (index(self) + 1))]
```

The second-order type for the method *top* is given explicitly here and enables parametric type inference on stacks. The argument x of the method *push* is typed with the special form *of(self)*, which means that the precondition (x ∈ of(self)) must be verified before applying the method.

This is a major improvement over the code presented in the first section of this paper. Not only do we get a more concise definition, but the condition.(x ∈ of(self)) can be checked automatically *by the compiler*. The key result for checking such statements at compile-time is the following [CP91]

$$[x]^* \leq \rho(p,[y]^*) \Rightarrow (x \in p(y))\ \textit{is always satisfied}$$

Thus, if the message *push*(s,x) is compiled, and if we know that s is a stack of integer and that x is an integer, the test $[x]^* \leq \rho(p,[y]^*)$ will be verified at compile-time and the run-time code will not include any type-checking. Thus we obtain the same efficiency as a statically type-checked language, with the flexibility of a dynamically type-checked language, since we can still have heterogeneous stacks.

[9] If the domain of the second method was *list*, the system will complain about a conflict on $\{()\} \times \{()\}$. This is another example that justifies the introduction of type difference.

6.4 Comparison with Related Works

This work is one among many attempts to introduce complex type inferences, such as those performed by ML [Mil78][HMM86], into an object-oriented language. Various type systems have been proposed to efficiently compile such languages [Wan87] [Red88] [CP89], but none of them was found to be adequate for LAURE [CP91] (Section 2). In fact, there are two families of approaches. The first tries to develop a statically strongly-typed language, such as Machiavelli [OBB89] or extended ML [HMM86]. Usually the problems that we presented in our motivations are solved, but there are restrictions on object-orientation or set operations. The second family [JGZ89] [CHC90] [PS91] tries to adapt more closely to object-oriented languages, but leaves the same problems open. We have found that the only way to escape this dilemma is to use second-order types. Another very important distinction in this family of work is the philosophy behind the objects. Most of the type-safe, completely static inference procedures that have been proposed recently use a notion of object that is more like a *value* than an object in the database sense (with real updates, side-effects and identity). For instance, although more powerful, the QUEST [Car89] type system (which subsumes our goals) could not be used for a language like LAURE.

Using second-order types for getting better type inference [Sce90][HM91] helped us to solve some very practical problems. The main difference with similar work comes from our order-sorted, extensible domain. What makes LAURE's type inference simpler is that abstract interpretation is only an approximation (and is still consistent [CC77]), which allows some simplification in the difficult cases, instead of running into complexity problems [HM91]. Using abstract interpretation to define type inference is a common idea, which is also applied to a simpler dynamically typed language in [AM91]. The reason we can afford to leave some difficult case out is that we use a dynamically typed language, which allows the compiler to generate type-ambiguous code if everything else failed. A last promising line of work is *safety analysis* [PA92], from which our work borrows some ideas.

Another distinction between the various works is the use of a set-based semantics, which offers many advantages from a data modeling perspective [SZ87] [LR89]. Although this is a controversial issue [CHC90], we believe that our model, based on sets and on a separation between taxonomy and methods, prevents some hard problems related in [PS91] to occur and provides a better foundation for method specialization and inheritance than [Mit90] in the case of an extensible and reflective object-oriented language.

7. Conclusion

In this paper we have shown how to extend methods in an object-oriented language into polymethods to solve some problems related to type-checking. Polymethods are extended into three directions: they are typed with a second-order type, which captures parametric and inclusion polymorphism in a complete and elegant manner; polymethods are attached directly to types instead of classes, which permits a finer description; last, some tests can be attached to methods' arguments that are verified at compile time by type inference. We have shown the benefits of a set-based type system for the LAURE language, from which we can derive a second-order type language and second-order type inference based on abstract interpretation.

The type system and its inference procedure have been implemented in the LAURE language, together with the second-order types. We have found them very practical since they allow type optimizations that used to be hard-coded into the compiler to be described in a formal and elegant manner. We are now implementing the second-order type inference procedure, to derive those types automatically (in the current version of the language, they are entered explicitly as a lambda expression from $\Lambda(T)$).

Acknowledgments

We would like to thank Guy Cousineau and Michel Bidoit for their comments on an earlier version of this paper. We are also grateful to all the contributors to the newsgroup *comp.object* who helped us in our thinking of a practical solution to the *min* problem. Last, we thank the anonymous referees for their detailed comments and suggestions.

References

[AM91] A. Aiken, B. R. Murphy: *Static Type Inference in a Dynamically Typed Language*. Proc. of the 18th Symposium on Principles of Programming Languages, 1991.

[AP90] H. Ait-Kaci, A. Podelski. *The Meaning of Life*. PRL Research Report, DEC, 1990.

[Car89] L. Cardelli: *Typeful Programming*. DEC SRC report #5, 1989.

[Ca89] Y. Caseau: "A Model For a Reflective Object-Oriented Language," *Sigplan Notices, Special Issue on Concurent Object-Oriented Programming,* 1989.

[Ca91] Y. Caseau: *An Object-Oriented Language for Advanced Applications*. Proc. of TOOLS USA 91, 1991.

[CC77] P. Cousot, R. Cousot: *Abstract Interpretation : A Unified Model for Static Analysis of Programs by Constructions or Approximation of Fixpoints.* Proc. Fourth ACM symposium of Principles of Programming Languages, 1977.

[CHC90] W. Cook, W. Hill, P. Canning: *Inheritance is not Subtyping.* Proc. of the 17th ACM Symposium on Principles of Programming Languages, San Francisco, 1990.

[CP89] W. Cook, J. Palsberg: *A Denotational Semantics of Inheritance and its Correctness*. Proc. of OOPSLA-89, New Orleans, 1989.

[CP91] Y. Caseau, L. Perron: *A type System for Object-Oriented Database Programming and Querying Languages.* Proc. of International Workshop on DataBase Programming Languages, 1991.

[CW85] L. Cardelli, P. Wegner: *On understanding types, data abstraction, and polymorphism.* ACM Computing Surveys, vol 17 n. 4, 1985.

[Don90] C. Dony: *Exception Handling and Object-Oriented Programming: Towards a Synthesis*. Proc. of OOPSLA'90, Ottawa, 1990.

[FW84] D.P. Friedman, M. Wand: *Reification: Reflection without Metaphysics.* ACM Symposium on LISP and Functional Programming, Austin, August 1984.

[GJ90] J. Graver, R. Johnson: *A Type System for Smalltalk.* Proc. of the 17th ACM Symposium on Principles of Programming Languages, San Francisco, 1990.

[GM86] J. Goguen, J. Meseguer: *Eqlog: Equality, Types and Generic Modules for Logic Programming*. In Logic Programming, Functions, Relations and Equations, Eds. D. DeGroot, G. Lindstrom, Prentice Hall, 1986.

[GMP90] L. Geurts, L. Meertens, S. Pemberton: *ABC Programmer's Handbook.* Prentice-Hall, 1990.

[GR83] A. Goldberg, D. Robson: *Smalltalk-80: The language and its implementation.* Addison-Wesley, 1983.

[HM91] F. Henglein, H. G. Mairson: *The Complexity of Type Inference for Higher-Order Typed Lambda Calculi.* Proc. of the 18th Symposium on Principles of Programming Languages, 1991.

[HMM86] R. Harper, D.B. MacQueen, R. Milner : *Standard ML.* Report ECS-LFCS-86-2, Dept. of Computer Science, University of Edimburgh, 1986.

[JGZ89] R. Johnson, J, Grauer, L. Zurawski: *TS: An Optimizing Compiler for Smalltalk.* OOPSLA-89, New Orleans, 1989.

[LR89] C. Lecluse, P. Richard: *Modelling Complex Structures in Object-Oriented Databases.* Proc. of ACM PODS, 1989.

[Me89] B. Meyer : *Static Typing for Eiffel.* Interactive Software Engineering, July, 1989.

[Mil78] R. Milner: *A theory of type polymorphism in programming.* In j. Computer and System Sciences vol 17 n. 3, 1978.

[Mit90] J. Mitchell: *Towards a Typed Foundation for Method Specialization and Inheritance.* Proc. of the 17th ACM Symposium on Principles of Programming Languages, San Francisco, 1990.

[MMM91] J. Mitchell, S. Meldal, N. Madhav: *An Extension of Standard ML modules with subtyping and inheritance.* Proc. of the 18th Symposium on Principles of Programming Languages, 1991.

[MN88] P. Maes, D. Nardi: *Meta-level Architecture and Reflection.* Elsevier Science Publication (North Holland), 1988.

[OB89] A. Ohori, P. Buneman: *Static Type Inference for Parametric Classes.* OOPSLA-89, New Orleans, 1989.

[OBB89] A. Ohori, P. Buneman, V. Breazu-Tannen: *Database Programming in Machiavelli - a Polymorphic Language with Static Type Inference.* ACM SIGMOD Conf. on Management of Data, May 1989.

[PS91] J. Palsberg, M. I. Schwartzbach: *Static Typing for Object-Oriented Programming.* DAIMI PB-355, June 1991.

[PS92] J. Palsberg, M. I. Schwartzbach: *Safety Analysis versus Type Inference for Partial Types.* DAIMI PB-404, July 1992.

[Red88] U. Reddy: *Objects as Closures: Abstract Semantics of Object-Oriented Languages.* Proc. ACM Conference on LISP and Functional Programming, 1988.

[Sce90] A. Scedrov: *A Guide to Polymorphic Types.* Logic and Computer Science, Academic Press, 1990.

[Smo89] G. Smolka: *Logic Programming over Polymorphically Order-Sorted Types.* PhD Thesis, Universität Kaiserslautern, May 1989.

[Str86] B. Stroustrup: *The C++ Programming Language.* Addison-Wesley, 1986.

[SZ87] A. Skarra, S. Zdonik: *Type Evolution in an Object-Oriented Database.* In Research Directions in Object Oriented Programming, ed. B. Schriver and P. Wegner, MIT press, 1987.

[Wan87] M. Wand: *Complete Type Inference for Simple Objects.* Proc. IEEE Symposium on Logic in Computer Science.