# Polymorphic Type Inference for a Simple Object Oriented Programming Language with State

Andreas V. Hense

Fachbereich 14

Universität des Saarlandes

Technischer Bericht Nr. A 20/90

# Polymorphic Type Inference for a Simple Object Oriented Programming Language with State

Andreas V. Hense

November 27, 1990

**Abstract**

We show how type inference for object oriented programming languages with state can be performed without type declarations. Our type inference system is based on the works of Rémy and Wand which can in turn be traced back directly to Milner's classical type inference algorithm.

O'small a simple object oriented language, is translated into $\mathcal{L}$, a language of $\lambda$-calculus with records and imperative features. Type inference rules are given for $\mathcal{L}$. O'small-programs are type checked after being translated into $\mathcal{L}$. We show that for translations of O'small-programs one can have a type system with principal types.

**keywords:** object oriented programming, polymorphic type inference, class inheritance, subtyping, imperative features

## 1 Introduction

The prominent object oriented languages with state are either statically typed with monomorphic types like Simula [DN66] or dynamically typed like Smalltalk [GR83]. There have been several approaches for type checking Smalltalk after the language definition had been completed. The problem is that either the language that was checked was not the whole Smalltalk [Suz81,BI82] or that some declarations still had to be introduced by the programmer [Gra89,GJ90].

The issue, whether object oriented programming languages need static type checking[1] or not, has been subject to vehement discussion in the past. Instead of making yet another contribution to this discussion we present a simple language with state where all types can be inferred. Because the programmer declares no types at all, the type checker can be seen as an optional device. A program that is refused by the type checker may be correct but one that is accepted cannot "go wrong". A type declaration possibility may still be useful for documentation but this is not the point of our consideration.

Section 2 introduces the language O'small and the type inference algorithm by examples. Methods that demand certain labels of their arguments can be applied to arguments with the required labels. The arguments may also have additional labels that are not required. The

[1]the terms *type checking* and *type inferencing* are used synonymously throughout this paper

consequence of this is something that resembles subtyping but there are differences. The way assignments may and may not be typed and the peculiarity that object oriented languages need types in the form of rational trees where other languages may not need them are the topic of two further examples.

Section 3 defines records and inheritance and thus prepares for section 4 where O'small is translated into $\lambda$-calculus with records and imperative features (a language called $\mathcal{L}$). Section 5 shows how we extend the type inference system of Wand by imperative constructs and some limitations of this approach. Section 6 concludes with the type inference rules for $\mathcal{L}$. O'small-programs are not type checked directly but via their translations to $\mathcal{L}$.

## 2 Example programs and their types

We introduce the object oriented programming language O'small by examples. For the definition of the language refer to [Hen90b]. Some properties of O'small are important for type checking: Parameters of methods are passed by reference but in the body of a method a formal parameter must not occur on the left hand side of an assignment. This is like "Small" [Gor79], the ancestor of O'small. A less common feature is that every variable has to be initialized with a value of the right type. This contrasts to many other languages where uninitialized variables have the value *nil*. There is no recursion in declarations. All recursion is achieved by fixed point construction, i.e. via sending messages to *self*. Classes are not first class objects in the language. Input and output have not been taken into consideration for the type checker. The treatment of output is trivial and the treatment of input is beyond the scope of this paper.

The types of the examples in the following section are a product of this article's type inferencer.

### 2.1 Row variables

The O'small program in figure 1 is about points and circles with Cartesian coordinates in the plane. Points and circles can be moved. There are two class definitions: *Point* inherits from *Base* and *Circle* from *Point*. The class *Base* is a class "without contents".

Objects of class *Point* have two instance variables representing the Cartesian coordinates of the point. A point object created with ***new*** is in the origin because its instance variables are initialized to zero. There are two methods for inspecting the instance variables because they are not directly visible from the outside. The method *move* changes the position of the receiver. In object oriented terminology the O'small expression *p.m(a)* stands for the sending of $m$ with argument $a$ to the receiver $p$. There is a method for the distance from the origin and a method that returns TRUE if the receiver is closer to the origin than the argument.

The class *Circle*, which inherits instance variables and methods from *Point*, has an additional instance variable for the radius, methods for reading and changing the radius, and it redefines *distFromOrg*. For the redefinition of *distFromOrg* the *distFromOrg*-definition of the superclass is referred to by *super.distFromOrg*.

The inherited function *closerToOrg* has not been redefined in the class *Circle*. In the body of *closerToOrg* the message *distFromOrg* is sent to *self*. If the receiver of a *closerToOrg*-message is a circle the redefined *distFromOrg*-method is chosen although *closerToOrg* has not been redefined.

```
class Point inheritsFrom Base
def var xComp := 0; var yComp := 0
in meth x() xComp
   meth y() yComp
   meth move(X,Y) xComp := X+xComp; yComp := Y+yComp
   meth distFromOrg()  sqrt(xComp*xComp + yComp*yComp)
   meth closerToOrg(point) self.distFromOrg < point.distFromOrg
ni

class Circle inheritsFrom Point
def var radius := 0
in meth r() radius
   meth setR(r) radius := r
   meth distFromOrg() max(0, super.distFromOrg - radius)
ni

def var p := new Point;
    var c := new Circle
in p.move(2,2); c.move(3,3); c.setR(2);
   output p.closerToOrg(c); {results in FALSE}
   p.move(0,-2);            c.move(0,-2);
   output p.closerToOrg(c)  {results still in FALSE}
ni
```

Figure 1: O'small program with points and circles

The output of example 1 results in: FALSE FALSE. This is what we intended. We are now able to compare points and circles with respect to their closeness to the origin and always get consistent behavior.

The types of the point object $p$ and the circle object $c$ of figure 1 are:

$$\begin{array}{lll} \text{p} & : [\text{closerToOrg} & : [\text{distFromOrg} : \text{num}, \mathcal{R}] \to \text{bool} \\ & \;\;\text{distFromOrg} & : \text{num} \\ & \;\;\text{move} & : \text{num} \to \text{num} \to \text{unit} \\ & \;\;\text{x} & : \text{num} \\ & \;\;\text{y} & : \text{num}] \end{array}$$

$$\begin{array}{lll} \text{c} & : [\text{closerToOrg} & : [\text{distFromOrg} : \text{num}, \mathcal{R}] \to \text{bool} \\ & \;\;\text{distFromOrg} & : \text{num} \\ & \;\;\text{move} & : \text{num} \to \text{num} \to \text{unit} \\ & \;\;\text{r} & : \text{num} \\ & \;\;\text{setR} & : \text{num} \to \text{unit} \\ & \;\;\text{x} & : \text{num} \\ & \;\;\text{y} & : \text{num}] \end{array}$$

Record types are denoted by their list of components with the component's types and optionally an ellipsis. An ellipsis is labeled by calligraphic letters (row variables (section 5.1)) and stands for the infinite set of labels that do not occur explicitly in the corresponding record type. The absence of an ellipsis means, the record type is not extendible. The type *unit* corresponds to the domain with one element. *unit* is the type of assignments. If a method ends with an assignment it has a type ending with *unit*; a method like this is more like a procedure. The types of methods appear in a curried version due to some technical detail of section 4.3.

## 2.2 Imperative features

Figure 2 shows how flexible the inferred types are with respect to variables and assignments.

Classes $B$ and $A$ define methods $m$ that demand of their argument $x$ to have a field with label

```
class A inheritsFrom Base meth m(x) x.h + x.i; x
class B inheritsFrom Base meth m(x) x.h + 1; x
class C inheritsFrom Base meth h()  0
class D inheritsFrom C    meth i()  0
class E inheritsFrom D    meth j()  0

def var a := new A     var b := new B
    var c := new C     var d := new D     var e := new E
in a.m(d); a.m(e);
   b.m(c);
   a := new B;
   a.m(d)
ni
```

Figure 2: O'small program with assignments

$h$ or fields with labels $h$ and $i$ respectively. The methods $m$ return their argument. Classes $C$, $D$, and $E$ are in an inheritance relation. They define methods with labels $h$, $i$, and $j$ and are thus possible arguments of the $m$-messages of objects of classes $A$ and $B$. But let us first look at the inferred types:

| | | | |
|---|---|---|---|
| a | : [m: [h : num, i : num, $\mathcal{R}$] | $\rightarrow$ | [h : num, i : num, $\mathcal{R}$]] |
| b | : [m: [h : num, $\mathcal{S}$] | $\rightarrow$ | [h : num, $\mathcal{S}$]] |
| c | : [h : num] | | |
| d | : [h : num, i : num] | | |
| e | : [h : num, i : num, j : num] | | |

The types of the objects reflect what has just been said about the classes. In particular the types of $c$, $d$, and $e$ "unfold" inheritance.

On the one hand this example shows that a method that demands certain fields of its argument can always be applied to an argument with additional fields: $d$ and $e$ can be arguments of $a.m$. One could argue that arguments of $a.m$ must be a subtype [CW85] of the type of $d$; but the notion of subtype does not appear in our context and the flexibility we can get with row variables does not always correspond to subtyping.

On the other hand the example shows that we have to be careful with assignments: $b.m(c)$ is type safe but $a.m(c)$ is not. If there were an assignment $b := a$ in the program $b.m(c)$ would not be type safe any more. The reader might argue that after the assignment $a := new\ B$ the argument $c$ becomes legal for $a.m$. In this particular program no type error would occur. But in the general case we are not sure if an assignment has taken place: the assignment may be part of an alternative of an if-expression. Also the distinction of textually before or after an assignment does not exclude any possible values in the general case: an occurrence of a variable can always be inside a method definition and the variable may be an instance variable.

When a variable and assignments to it are type checked, all possible values have to be taken into account – at all occurrences of the variable. The way assignments are dealt with can be seen in more detail in section 6. Had we assigned $b$ to $a$ instead of *new* $B$, the program would not have passed the type checker. This is due to the fact that $b$ on the right hand side of the assignment is an occurrence of the variable $b$ and consequently its type does not contain any generic variables. For a detailed description refer to [Hen91]. Although the types of the objects $a$ and $b$ are not in any subtype relation one can assign objects of class $B$ to a variable containing objects of class $A$.

## 2.3 Recursive types

Figure 3 is a natural example[2] and demonstrates the need for recursive types. The objects of class *Pair* together with the relation *leq* define a preorder. The objects of class *OrderedPair*

```
class Pair inheritsFrom Base
def var xComp:=0  var yComp:=0
in meth set(a,b) xComp := a; yComp := b
   meth x()      xComp
   meth y()      yComp
   meth leq(p)   (xComp+yComp) <= (p.x+p.y)
   meth eq(p)    xComp=p.x and yComp=p.y
ni

class OrderedPair inheritsFrom Pair
   meth eq(e)    self.leq(e) and e.leq(self)

def var p  := new OrderedPair in p.set(7,3) ni
```

Figure 3: O'small program with recursive types

together with the relation *leq* and the equality *eq* define a partial order because the equality has been redefined and now *leq* becomes antisymmetric. The type $\sigma$ of the object $p$ is circular and defined by the following equation:

[2]a modification of an example in [Hen90a]

$$\begin{array}{lll}\sigma & = [\text{eq}: [\text{leq}: \sigma \rightarrow \text{bool}, \text{x}: \text{num}, \text{y}: \text{num}, \mathcal{R}] & \rightarrow \text{bool} \\ & \phantom{=[}\text{leq}: [\text{leq}: \sigma \rightarrow \text{bool}, \text{x}: \text{num}, \text{y}: \text{num}, \mathcal{R}] & \rightarrow \text{bool} \\ & \phantom{=[}\text{set}: \text{num} \rightarrow \text{num} \rightarrow \text{unit} & \\ & \phantom{=[}\text{x}\ : \text{num} & \\ & \phantom{=[}\text{y}\ : \text{num}] & \end{array}$$

The recursion in the type of $\sigma$ stems from the redefinition of the equality in the class *OrderedPair*. The argument *e* of *self.leq* must understand a message *leq* where the same *self* is an argument.

The perspicacious reader may argue that *self.leq* demands an $x-$ and a $y$-component of its argument and not a *leq*-component. However this knowledge can only be gained if one looks at the definition of the method *leq* in the class *Pair*. Method definitions are not (mutually) recursive, whence type checking does not regard other methods in a class. The other methods are therefore not known in advance (this corresponds to 'bound by $\lambda$'). If methods were known in advance this would correspond to 'bound by *let*'. In the $\lambda$-case we have non generic variables and in the *let*-case we have generic variables. Because *e* and *self* are bound by $\lambda$, *self.leq* must have the type of its argument on the left hand side of the arrow: *self.leq: type(e)* $\rightarrow$ *bool.*

The verification whether the type of *self* and the type of the actually provided record of methods can be unified is done at object creation time. It is possible to send messages to *self* that are not defined in the class. This results in an abstract class [GR83]. The type checker accepts abstract classes but rejects the creation of their objects.

## 3 Basic definitions

This section consists of definitions for inheritance taken from [Coo89]. Here objects have no state. Therefore it is not directly a description of O'small but these definitions are the basis for the denotational semantics of O'small [Hen90b]. Furthermore some of the definitions are used in section 4.3.

**Definition 1** A *record* is a finite mapping from a set of labels into a set of values. A record is denoted by $\begin{bmatrix} x_1 & \mapsto & v_1 \\ & \vdots & \\ x_n & \mapsto & v_n \end{bmatrix}$ with labels $x_i$ and values $v_i$. All labels that are not in the list are mapped onto *absent*. The empty record, where all labels are mapped onto *absent* is denoted by $[\,]$.

**Definition 2** Let $dom(m) = \{x \mid m(x) \neq absent\}$. The *left-preferential combination of records* is defined by:

$$(m \oplus n)(s) = \begin{cases} m(s) & \text{if } s \in dom(m) \\ n(s) & \text{if } s \in dom(n) - dom(m) \\ absent & \text{otherwise} \end{cases}$$

An *object* is a record with functions (methods) as values. A *generator* is a function to which a fixed point operator can be applied. Its first formal parameter represents self-reference. A *class* is a generator that creates objects by fixed point operation. The domain of classes is Class = Object $\rightarrow$ Object. *Inheritance* is the derivation of a new generator from an existing one, where the formal parameters for self-reference of both generators are shared. A *wrapper* is a function that modifies a generator in a self-referential way. A wrapper has a parameter for self-reference and a parameter for the generator it modifies. Definition 3 and definition 4 will show how wrappers operate on classes:

**Definition 3** Let $*$ be a binary operator on values. The *distributive version* of $*$ is denoted by $\boxed{*}$. It operates on generators and is defined by:

$$G_1 \boxed{*} G_2 = \lambda s.G_1(s) * G_2(s)$$

**Definition 4** The *inheritance function* $\boxed{\triangleright}$ applies a wrapper $W$ to a class $C$ and returns a class. $\triangleright$ is defined by: $w \triangleright c = (w \cdot c) \oplus c = w(c) \oplus c$ and $\cdot$ is the application. Hence:

$$W \boxed{\triangleright} C = (W \boxed{\cdot} C) \boxed{\oplus} C$$

From these definitions follows that the domain of wrappers is Wrapper = Object $\rightarrow$ Object $\rightarrow$ Object. Wrappers are central to the semantics of inheritance. In every class declaration a superclass is named. If nothing is inherited *Base* is named as superclass. *Base* is the class whose objects are empty records. The semantics of a class definition is a wrapper being wrapped around the superclass, and this results in a new class. As pointed out above we can get objects by applying the fixed point operator to a class. Note that in this context classes have no additional parameters and thus the objects of a class would be all equal, if they had no state.

## 4 Translation into $\lambda$-calculus with imperative features

We proceed by translating O'small into $\lambda$-calculus with imperative features and call this language $\mathcal{L}$. The type inference works for a subset of $\mathcal{L}$-programs. The type safeness of the translation of an O'small-program implies type safeness of the original program.

### 4.1 Syntax of O'small

There are primitive syntactic domains,

| | | |
|---|---|---|
| Ide | the domain of identifiers | I |
| Bas | the domain of basic constants | B |
| BinOp | the domain of binary operators | O |

compound syntactic domains,

| | | |
|---|---|---|
| Pro | the domain of programs | P |
| Exp | the domain of expressions | E |
| CExp | the domain of compound expressions | C |
| Var | the domain of variable declarations | V |
| Cla | the domain of class declarations | K |
| Meth | the domain of method declarations | M |

and syntactic clauses:

| | |
|---|---|
| P | ::= K C |
| K | ::= class $I_1$ inheritsFrom $I_2$ def V in M \| $K_1$ $K_2$ \| $\epsilon$ |
| C | ::= E \| I := E \| if E then $C_1$ else $C_2$ \| def V in C \| $C_1$;$C_2$ |
| E | ::= B \| I \| E.I($E_1$,...,$E_n$) \| new E \| $E_1$ O $E_2$ |
| V | ::= var I := E \| $V_1$ $V_2$ \| $\epsilon$ |
| M | ::= meth I($I_1$,...,$I_n$) C \| $M_1$ $M_2$ \| $\epsilon$ |

Input and output are not considered. The syntax is abstract: closing parentheses like *ni* do not figure here. The semantics of O'small is described formally in [Hen90b].

## 4.2 Syntax of $\mathcal{L}$

The primitive syntactic domains of $\mathcal{L}$ are the same as the primitive syntactic domains of O'small. In $\mathcal{L}$ a formula is also a program.

| | | |
|---|---|---|
| For | the domain of formulas | F |
| Rec | the domain of records | R |

The syntactic clauses:

| | | |
|---|---|---|
| F | ::= | I \| B \| R \| $F_1$ O $F_2$ \| $F_1$ $F_2$ \| if $F_1$ then $F_2$ else $F_3$ \| $\lambda$I.F \| **Y** F \| |
| | | let I = $F_1$ in $F_2$ \| def I = $F_1$ in $F_2$ \| I := F |
| R | ::= | [] \| R with I = F \| $R_1 \oplus R_2$ \| R.I |

The semantics of the language $\mathcal{L}$ is $\lambda$ calculus with imperative features. **Y** is the fixed point operator. *let* declares a value and *def* declares a variable. Variables may appear on the left hand side of assignments.

## 4.3 The translation function

The translation function $\langle\langle\_\rangle\rangle$ : O'small $\rightarrow \mathcal{L}$ yields a subset of $\mathcal{L}$.[3] We have $codomain(\langle\langle\_\rangle\rangle) \subset \mathcal{L}' \subset \mathcal{L}$. The type inference of section 6 is valid for $\mathcal{L}$' and not for $\mathcal{L}$. The syntax of $\mathcal{L}$' will be given in [Hen91]. In $\mathcal{L}$' we can assume that for every record concatenation $r_1 \oplus r_2$ the $r_i$ $(i \in 1,2)$ are "known". I.e. $r_i$ is either a constant record or a variable bound by *let*. It is above all never a variable bound by $\lambda$. With this restriction we are able to do a simpler type checking than [Wan89b] and we have principal types (see also section 5.2).

The translation of primitives is not listed here:

[3] _ is a place holder

$$\begin{array}{lcl}
\langle\!\langle \mathrm{K}_1 \ldots \mathrm{K}_n \; \mathrm{C} \rangle\!\rangle & = & \mathrm{let} \; \langle\!\langle \mathrm{K}_1 \rangle\!\rangle \; \mathrm{in} \ldots \mathrm{let} \; \langle\!\langle \mathrm{K}_n \rangle\!\rangle \; \mathrm{in} \; \langle\!\langle \mathrm{C} \rangle\!\rangle \\
\langle\!\langle \mathrm{class} \; \mathrm{I}_1 \; \mathrm{inheritsFrom} \; \mathrm{I}_2 \; \mathrm{def} \; \mathrm{V} \; \mathrm{in} \; \mathrm{M} \rangle\!\rangle & = & \mathrm{I}_1 = (\lambda \mathrm{self}.\lambda \mathrm{super}.\langle\!\langle \mathrm{def} \; \mathrm{V} \; \mathrm{in} \; \mathrm{M} \rangle\!\rangle) \; \boxed{\triangleright} \; \langle\!\langle \mathrm{I}_2 \rangle\!\rangle \\
\langle\!\langle \mathrm{def} \; \mathrm{V}_1 \ldots \mathrm{V}_n \; \mathrm{in} \; \mathrm{M} \rangle\!\rangle & = & \mathrm{def} \; \langle\!\langle \mathrm{V}_1 \rangle\!\rangle \; \mathrm{in} \ldots \mathrm{def} \; \langle\!\langle \mathrm{V}_n \rangle\!\rangle \; \mathrm{in} \; \langle\!\langle \mathrm{M} \rangle\!\rangle \\
\langle\!\langle \mathrm{var} \; \mathrm{I} := \mathrm{E} \rangle\!\rangle & = & \langle\!\langle \mathrm{I} \rangle\!\rangle = \langle\!\langle \mathrm{E} \rangle\!\rangle \\
\langle\!\langle \mathrm{M}_1 \ldots \mathrm{M}_n \rangle\!\rangle & = & [] \; \mathrm{with} \; \langle\!\langle \mathrm{M}_1 \rangle\!\rangle \ldots \mathrm{with} \; \langle\!\langle \mathrm{M}_n \rangle\!\rangle \\
\langle\!\langle \mathrm{meth} \; \mathrm{I}(\mathrm{I}_1 \ldots \mathrm{I}_n) \; \mathrm{C} \rangle\!\rangle & = & \langle\!\langle \mathrm{I} \rangle\!\rangle = \lambda \; \langle\!\langle \mathrm{I}_1 \rangle\!\rangle. \; \ldots \lambda \; \langle\!\langle \mathrm{I}_n \rangle\!\rangle. \; \langle\!\langle \mathrm{C} \rangle\!\rangle \\
\langle\!\langle \mathrm{C}_1; \ldots ; \mathrm{C}_n \rangle\!\rangle & = & \mathrm{let} \; \mathrm{newVar} = \langle\!\langle \mathrm{C}_1 \rangle\!\rangle \; \mathrm{in} \; \langle\!\langle \mathrm{C}_2; \ldots ; \mathrm{C}_n \rangle\!\rangle \\
\langle\!\langle \mathrm{I} := \mathrm{E} \rangle\!\rangle & = & \langle\!\langle \mathrm{I} \rangle\!\rangle := \langle\!\langle \mathrm{E} \rangle\!\rangle \\
\langle\!\langle \mathrm{if} \; \mathrm{E} \; \mathrm{then} \; \mathrm{C}_1 \; \mathrm{else} \; \mathrm{C}_2 \rangle\!\rangle & = & \mathrm{if} \; \langle\!\langle \mathrm{E} \rangle\!\rangle \; \mathrm{then} \; \langle\!\langle \mathrm{C}_1 \rangle\!\rangle \; \mathrm{else} \; \langle\!\langle \mathrm{C}_2 \rangle\!\rangle \\
\langle\!\langle \mathrm{def} \; \mathrm{V}_1 \ldots \mathrm{V}_n \; \mathrm{in} \; \mathrm{C} \rangle\!\rangle & = & \mathrm{def} \; \langle\!\langle \mathrm{V}_1 \rangle\!\rangle \; \mathrm{in} \ldots \mathrm{def} \; \langle\!\langle \mathrm{V}_n \rangle\!\rangle \; \mathrm{in} \; \langle\!\langle \mathrm{C} \rangle\!\rangle \\
\langle\!\langle \mathrm{new} \; \mathrm{E} \rangle\!\rangle & = & \mathbf{Y} \; \langle\!\langle \mathrm{E} \rangle\!\rangle \\
\langle\!\langle \mathrm{E}_1 \; \mathrm{O} \; \mathrm{E}_2 \rangle\!\rangle & = & \langle\!\langle \mathrm{E}_1 \rangle\!\rangle \; \langle\!\langle \mathrm{O} \rangle\!\rangle \; \langle\!\langle \mathrm{E}_2 \rangle\!\rangle \\
\langle\!\langle \mathrm{E.I}(\mathrm{E}_1 \ldots \mathrm{E}_n) \rangle\!\rangle & = & (\langle\!\langle \mathrm{E} \rangle\!\rangle.\langle\!\langle \mathrm{I} \rangle\!\rangle) \; \langle\!\langle \mathrm{E}_1 \rangle\!\rangle \ldots \langle\!\langle \mathrm{E}_n \rangle\!\rangle
\end{array}$$

The expression before the inheritance operator in the second clause is a wrapper. *self* and *super* are $\lambda$-abstracted; they have the same meaning as in Smalltalk. When methods are translated they are curried. Correspondingly message sends (record selection) are translated into a curried version. *newVar* is a distinct new variable for every $C_i$ that is translated; it is a dummy variable that cannot be used.

The denotational semantics of O'small is contained in [Hen90b]. The translation into $\mathcal{L}$ does not just serve type checking but is another possibility of defining the semantics of O'small; all that remains to be defined after the translation is a denotational semantics of $\mathcal{L}$.

## 5 Development of the type inference algorithm

### 5.1 The inference system of Wand

[Wan89b] considers type inference for the $\lambda$-calculus with records. Wand extends the approach of [Rem89] by record concatenation and unbounded label sets. Thus a reduction of the type checking of an object oriented language without state to the classical type checking of [Mil78] is achieved. Correctness proofs of [Mil78] extend to this new approach. Objects, classes, and inheritance are dealt with in about the same way as in section 4, i.e. by translating them into $\lambda$-terms. The difference is that here we have imperative features.

The type of a record is the record of the types of its components. Thus the type of a record is a function $L \rightarrow (Type + absent)$ where $L$ is the set of labels. The type constructors may look like this in a first approach:

$$\begin{array}{rcl}
\rightarrow & : & Type \times Type \Rightarrow Type \\
\prod & : & [L \Rightarrow (Type + absent)] \Rightarrow Type
\end{array}$$

where $\Rightarrow$ is a kind constructor. But this is no ordinary algebraic signature because $L$ is infinite. [Rem89] turned this into an ordinary algebraic signature and [Wan89b] made the extension to infinite label sets, so that we get the type constructors:

$$\begin{array}{rcl} \rightarrow & : & Type \times Type \Rightarrow Type \\ \prod & : & Field^n \times Extension \Rightarrow Type \\ absent & : & Field \\ pres & : & Type \Rightarrow Field \\ empty & : & Extension \end{array}$$

$n$ is the number of labels actually appearing in the program, so $\prod$ is finite. Unbounded label sets are represented finitely by a finite number of explicit labels and and an extension. An empty extension stands for infinitely many labels with absent fields and an extension variable (row variable) stands for infinitely many labels with fields that can be either absent or present. Labels are implicit in this notation (position in a tuple). Record constants have principal types:

$$\begin{array}{rcl} [] & : & \prod[\underbrace{absent, \ldots, absent}_{n}, empty] \\ \_.a & : & \prod[f_1, \ldots, pres(t), \ldots, f_n, \mathcal{R}] \rightarrow t \\ \_\ with\ a = \ \_ & : & \prod[f_1, \ldots, f_a, \ldots, f_n, \mathcal{R}] \rightarrow t \rightarrow \prod[f_1, \ldots, pres(t), \ldots, f_n, \mathcal{R}] \end{array}$$

The $f_i$ are variables of kind field. Extension variables (row variables) are denoted by calligraphic upper case letters. In $\_.a$ the underlining is a place holder for a record argument and the dot stands for record selection. In $\_$ *with* $a =$ $\_$ the first underlining is a place holder for the record to be updated and the second underlining is a place holder for the new value. $f_a$ indicates that if a record is updated at a label $a$ the field may have been either present or absent before.

In practice when one incrementally checks a program one does not know the set of labels in advance. For this case [Wan89b] presents a technique called padding of which we only describe the main idea here (for details refer to [Wan89b,Wan89a]). Every record type contains only the previously encountered labels explicitly. When two record types are to be unified they are padded such that the union of their explicit labels is now explicit in both record types and that new row variables now range over the remaining implicit labels. E.g.: $r_1 : [a : \sigma, b : \tau, \mathcal{R}]$ and $r_2 : [b : \tau', c : \upsilon', \mathcal{S}]$ are padded. This results in $r_1' : [a : \sigma, b : \tau, c : \alpha, \mathcal{R}']$ and $r_2' : [a : \beta, b : \tau', c : \upsilon', \mathcal{S}']$ where the row variables $\mathcal{R}$ and $\mathcal{S}$ have to be substituted accordingly. The padded types can be unified field by field.

## 5.2 Principal types

O'small does not have classes as first class values. The full generality of Wand's type inference system, where there are no principal types, is not necessary. The absense of principal types can be made plausible by the formula $\lambda x.\lambda y.(x \oplus y).a$. One ignores whether the label $a$ must occur in $x$ or in $y$. The record concatenation operator $\_ \oplus \_$ has no principal type in general. In the translation function (section 4.3) $\_ \oplus \_$ appears in the second clause: it is hidden inside the inheritance operator $\_ \boxed{\triangleright} \_$.

$M$, the record appearing on the left hand side of $\boxed{\triangleright}$ in the translation function, is translated by a finite sequence of *with*'s. By a simple calculation we can see that $\oplus$ can be eliminated and an replaced by a finite sequence of *with*'s in the second clause. Now we are back to record constants and they have principal types according to section 5.1.

## 5.3 Imperative constructs

This section and section 5.4 assume detailed knowledge of the type checking algorithm of [Mil78].

Most literature about type inference is concerned with purely applicative versions of object oriented languages. One essential feature of object oriented languages is state [Weg87]. Existing solutions to the problem of references to the store are either restrictive (top-level references must be monomorphic [GMW79]) or they lead to an intricate algorithm [Dam85,Tof88].

Before we continue with the idea of our solution let us consider the special case of an assignment of one variable to another like $x := y$, and the trap one may fall into when devising type checking for the assignment statement. At first glance it may seem enough to unify the types of the two variables. This may not lead to the most flexible of type systems but should be safe. Simple unification goes wrong because of let-polymorphism [Mil78]: as $x$ and $y$ occur in the assignment, only instances of their generic types are unified. Thus the generic type of $x$ is not influenced.

**Example 1** Let $x$, $y$, and $z$ be objects having the following types with generic row variables $\mathcal{R}$ and $\mathcal{S}$:[4]

$$\begin{aligned} x &: [a : [a : num, b : num, \mathcal{R}] \rightarrow [a : num, b : num, \mathcal{R}]] \\ y &: [a : [b : num, \mathcal{S}] \rightarrow [b : num, \mathcal{S}]] \\ z &: [b : num] \end{aligned}$$

Consider the program fragment *y.a(z); y:=x; y.a(z)*. If the row variables $\mathcal{R}$ and $\mathcal{S}$ are generic, the unification of their instances does not influence the generic types. The program would be considered correct although the last expression produces a type error.

If we restrict references to nongeneric types we lose all our polymorphism. The idea to the solution is the following. All assignments to a variable occur in its scope. If a variable is an actual parameter in a selected method (in object oriented terminology a parameter in a message) it cannot be assigned in the method because formal parameters must not appear on the left hand side of assignments. An imperative variable is declared by $def\ x := e_0\ in\ s\ ni$ and in its scope (inside $s$) there are assignments of the form $x := e_i \ \ (1 \leq i \leq n)$. As we are not concerned with flow analysis here we assume that at any occurrence of $x$ its value may be an instance of any of the generic types of $e_0$ or $e_i$ (the possible types). Thus for every occurrence of $x$ we unify instances of all possible types and thus get a type for this occurrence. Note that, algorithmically speaking, for unifying all instances of possible types the unification has to be delayed until the end of the scope of $x$. All type information has to be kept in a table (definition 7). With this procedure we get generic reference types in many cases. In particular, it is possible to have full flexibility and generic types for a variable with no assignments to it – without any distinction between constants and variables.

## 5.4 Limitations

There are limitations to our type system. The program of figure 4 which is obviously correct is refused by the type checker. The implicit subtyping mechanism that works for arguments of methods (see figure 2) does not work for objects that are known "in advance" because these objects have non-extendible record types and the different possible types are unified in our

[4]here labels have names

algorithm. It is not possible to give extendible record types to objects because then, every record selection would be accepted by the type checker.

Again in the general case one is not sure if the assignment has taken place but if the only message sent to $a$ is $m$ without arguments this cannot go wrong. In this example objects of the subclass $B$ are of a subtype of the types of objects of the superclass $A$ – i.e. if we define a subtype relation appropriately. The problem is that we do not know of any simple way of integrating an explicit subtype notion into the current framework. [Sta88] may be valuable in this direction of research.

One possible way out of this problem is a modification of the treatment of imperative tables.

```
class A inheritsFrom Base meth m() 0
class B inheritsFrom A    meth n() 0

def var a := new A   var b := new B
in a := b; a.m ni
```

Figure 4: O'small program with inextendible types

If instead of unifying instances of all possible types we start a continuation of the type checking for each possibility, the above example goes through but the number of possible types grows exponentially with the length of the program in the worst case. That clearly makes this solution an impasse. Note that in the framework of Milner's algorithm and thus in our framework there is no explicit notion of subtypes. A statement like $\tau$ is a subtype of $\sigma$ is therefore impossible.

## 6 Type inference rules

For type inference rules we follow the notation used in [Rem89]. Let $T$ denote the set of types (in [Rem89] they are called sorts); types are first order kinded rational trees over the set of variables $V = V^t \cup V^f$ and the set of symbols $S = C \cup B$, where $C = \{\rightarrow, \prod, absent, pres, empty\}$, $B = \{num, bool, unit\}$ is the set of basic types, $V^t$ the set of type variables, and $V^f$ the set of field variables.

Type variables are denoted by letters $\alpha$, $\beta$, $\gamma$; they have a subscript $_g$ if they are generic. Types are denoted by letters $\sigma$, $\tau$; they have a subscript $_g$ if generic types are admitted. A type is generic if it may contain generic variables. $\langle A\rangle$ is the set of all type variables occurring in $A$.

**Definition 5** A *graft* is a mapping from $V$ to $T$. We only use finite grafts. A graft $\mu$ is finite if $\{\alpha \mid \alpha\mu \neq \alpha\}$ is finite. Grafts are denoted by letters $\mu$, $\nu$.

Rational trees can be regarded as the application of a finite graft to a variable $\epsilon$, whence they are finitely representable.

**Definition 6** A generic type $\tau_g$ is a *generic instance* of a generic type $\sigma_g$ iff there exists a graft $\mu$ such that $\tau_g = \sigma_g\mu$.

What substitutions are for terms, grafts [Hue76] are for rational trees.

$$(TAU1)\quad \frac{}{A \vdash I : \sigma_g} \qquad (I : \sigma_g \in A)$$

$$(TAU2)\quad \frac{}{A \vdash B : num}$$

$$(VAR)\quad \frac{}{A \vdash I : \sigma} \qquad (I : \Sigma \in A, \Sigma \downarrow \sigma)$$

$$(INST)\quad \frac{A \vdash F : \sigma_g}{A \vdash F : \sigma_g \mu_g}$$

$$(GEN)\quad \frac{A \vdash F : \sigma_g}{A \vdash F : \sigma_g[\alpha_g/\alpha]} \qquad (\alpha \notin \langle A \rangle)$$

$$(ABS)\quad \frac{A_I \cup \{I : \sigma\} \vdash F : \tau}{A \vdash \lambda I.F : \sigma \rightarrow \tau}$$

$$(APP)\quad \frac{A \vdash F_1 : \sigma \rightarrow \tau \qquad A \vdash F_2 : \sigma}{A \vdash (F_1\ F_2) : \tau}$$

$$(IF)\quad \frac{A \vdash F_1 : bool \qquad A \vdash F_2 : \tau \qquad A \vdash F_3 : \tau}{A \vdash if\ F_1\ then\ F_2\ else\ F_3 : \tau}$$

$$(FIX)\quad \frac{A \vdash F : \tau \rightarrow \tau}{A \vdash \mathbf{Y}\ F : \tau}$$

$$(LET)\quad \frac{A \vdash F_1 : \sigma_g \qquad A_I \cup \{I : \sigma_g\} \vdash F_2 : \tau}{A \vdash let\ I = F_1\ in\ F_2 : \tau}$$

$$(DEF)\quad \frac{A \vdash F_1 : \sigma_g \qquad A_I \cup \{I : \Sigma\} \vdash F_2 : \tau}{A \vdash def\ I = F_1\ in\ F_2 : \tau} \qquad (\sigma_g \in \Sigma)$$

$$(ASS)\quad \frac{A \vdash F : \sigma_g}{A \vdash I := F : unit} \qquad (I : \Sigma \in A, \sigma_g \in \Sigma)$$

Figure 5: Type inference rules

**Definition 7** An *imperative table* $\Sigma$ is a finite set of generic types. A type $\sigma$ is a *common instance* of an imperative table $\Sigma = \{\sigma_{g1}, \ldots, \sigma_{gn}\}$ (in symbols: $\Sigma \downarrow \sigma$) iff

$$\forall i \in \{1, \ldots, n\}\ \exists \mu_i : \quad \sigma_{gi}\mu_i = \sigma$$

The type inference rules of figure 5 are applied to triples $(A, F, \tau_g)$ denoted by $A \vdash F : \tau_g$. $A$ is the environment of assumptions of the form $x : \sigma_g$. $A_x$ denotes all assumptions in $A$ except those for variable $x$. $F$ is a formula of $\mathcal{L}'$.

Type inference does not start with the empty environment of assumptions but with the environment $\{Base : \alpha_g \rightarrow []\}$. This is the type of the predefined Base-class: on the right hand side the type of the empty record and on the left hand side any type for self reference.

The inference rules are in figure 5. Let us explain the "imperative" inference rules: When an assignable variable is encountered (rule VAR) its type must be a common instance of the possible types in $\Sigma$. When an assignable variable is declared (rule DEF) a new imperative table $\Sigma$ that contains $\sigma_g$ is constructed. The other generic types in $\Sigma$ should be those that are inferred on the right hand side of assignments to $I$. When a variable is assigned a value (rule ASS) we check whether the type of the right hand side is inside the possible types of $I$.

The algorithm has an effective way of constructing the imperative tables and does not have to guess. In the algorithm the checking of imperative tables is delayed until the end of the scope of a variable because only then we know all possible types and all occurrences of the identifier.

## 7 Conclusion

We have extended Wand's algorithm to languages with imperative features. References with generic types are realized by delayed unification. This technique may be applicable to other language classes as well.

A similar algorithm but without imperative features can be found in [BSW90].

Our approach is still relatively simple and elegant – mainly owing to the work that has been done before – and shows unexpected flexibility in many respects. There are however some severe restrictions (figure 4) which seem to be inherent to all type inferencers in this line of research. A remedy may lie in an explicit subtype notion and least upper bounds in the lattice of types [Sta88].

The type inference is easily extended to an object oriented language with explicit wrappers [Hen90a]. This language is a slight modification of O'small and has multiple inheritance. For a language with explicit wrappers the documentary effect of type inference is even more important.

The types of objects of subclasses are not in a subtype relation to the types of objects of their superclasses [CHC90]. The test whether a class is abstract comes as a gift with this type inferencer.

Formal proofs of several correctness issues and the type inference algorithm will be contained in the author's forthcoming Ph.D. thesis [Hen91]. The algorithm has been implemented in ML [Mil85,HMM86] and uses ML-Yacc [TA90].

**Acknowledgements**

I am grateful to Reinhold Heckmann, Fritz Müller, Georg Sander, Gert Smolka, and Reinhard Wilhelm for illuminating discussions and comments on drafts of this paper.

## References

[BI82] Alan H. Borning and Daniel H. H. Ingalls. A type declaration and inference system for Smalltalk. In *Symposium on Principles of Programming Languages*, pages 133–139, ACM, 1982.

[BSW90] M. Beaven, R. Stansifer, and D. Wetklow. A statically-typed language with classes. February 1990. Purdue University.

[CHC90] W. Cook, W. Hill, and P. Canning. Inheritance is not subtyping. In *Symposium on Principles of Programming Languages*, pages 125–135, ACM, San Francisco, January 1990.

[Coo89] William R. Cook. *A Denotational Semantics of Inheritance*. Technical Report CS-89-33, Brown University, Dept. of Computer Science, Providence, Rhode Island 02912, May 1989.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[Dam85] L. Damas. *Type Assignment in programming languages*. PhD thesis, University of Edinburgh, 1985. CST-33-85.

[DN66] O-J. Dahl and K. Nygaard. Simula - an Algol-based simulation language. *Communications of the ACM*, 9:671–678, 1966.

[GJ90] Justin O. Graver and Ralph E. Johnson. A type system for Smalltalk. In *Symposium on Principles of Programming Languages*, pages 136–150, ACM, San Francisco, January 1990.

[GMW79] M. Gordon, R. Milner, and C. Wadsworth. Edinburgh LCF. *Lecture Notes in Computer Science*, 78, 1979.

[Gor79] M.J.C. Gordon. *The Denotational Description of Programming Languages: An Introduction*. Springer-Verlag, New York/Heidelberg/Berlin, 1979.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: the Language and its Implementation*. Addison-Wesley, 1983. revised in 1989.

[Gra89] Justin O. Graver. *Type-Checking and Type Inference for Object-Oriented Programming Languages*. PhD thesis, University of Illinois at Urbana-Champaign, 1989.

[Hen90a] Andreas V. Hense. *Denotational Semantics of an Object Oriented Programming Language with Explicit Wrappers*. Technical Report A 11/90, Universität des Saarlandes, Fachbereich 14, 1990.

[Hen90b] Andreas V. Hense. *Wrapper Semantics of an Object Oriented Programming Language with State*. Technical Report A 14/90, Universität des Saarlandes, Fachbereich 14, 1990.

[Hen91] Andreas V. Hense. *Polymorphic type checking for object oriented programming languages*. PhD thesis, Universität des Saarlandes, Fachbereich 14, D-6600 Saarbrücken, 1991. in preparation.

[HMM86] Robert Harper, David MacQueen, and Robin Milner. Standard ML. In *Laboratory for Foundations of Computer Science Report Series*, March 1986.

[Hue76] Gérard Huet. *Résolution d'équations dans les langages d'ordre 1,2,...,$\omega$*. PhD thesis, Université Paris 7, 1976. Thèse de doctorat d'état.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978.

[Mil85] Robin Milner. The standard ML core language. In *Polymorphism The ML/LCF/Hope Newsletter*, 1985. Vol. II, No.2.

[Rem89] Didier Rémy. Typechecking records and variants in a natural extension of ML. In *Symposium on Principles of Programming Languages*, pages 77–88, ACM, 1989.

[Sta88] Ryan Stansifer. Type inference with subtypes. In *Symposium on Principles of Programming Languages*, pages 88–97, ACM, January 1988.

[Suz81] Norihisa Suzuki. Inferring types in Smalltalk. In *Symposium on Principles of Programming Languages*, ACM, January 1981.

[TA90] David R. Tarditi and Andrew W. Appel. *ML-Yacc, version 2.0, Documentation for release version*. Carnegie Mellon University, April 1990.

[Tof88] Mads Tofte. *Operational Semantics and Polymorphic Type Inference*. PhD thesis, University of Edinburgh, May 1988. CST-52-88 also published as ECS-LFCS-88-54.

[Wan89a] Mitchell Wand. *Type Inference for objects with instance variables and inheritance*. Technical Report NU-CCS-89-2, Northeastern University, Boston, 1989.

[Wan89b] Mitchell Wand. Type inference for record concatenation and multiple inheritance. In *Logic in Computer Science*, pages 92–97, 1989.

[Weg87] Peter Wegner. The object-oriented classification paradigm. In B. Shriver and P. Wegner, editors, *Research Directions in Object Oriented Programming*, pages 479–560, MIT Press, 1987.