Type Inference

for

O’small

Andreas V. Hense
Fachbereich 14
Universität des Saarlandes
Technischer Bericht Nr. A 06/91

LIMITED DISTRIBUTION NOTICE
This report has been submitted for publication and will probably be copyrighted if accepted. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the publisher, its distribution prior to publication should be limited to peer communication and specific requests.
Type Inference for O’SMALL

Andreas V. Hense

Universität des Saarlandes
Im Stadtwald 15
6600 Saarbrücken 11, Germany
hense@cs.uni-sb.de

October 23, 1991

Abstract

Type inference for the $\lambda$-calculus with records was investigated by Rémy and also by Wand. In this paper we specialize Wand’s approach in order to obtain principal types. We develop a novel technique for the treatment of imperative constructs that fits well with Wand’s type inference. The result is type inference for a $\lambda$-calculus with records and imperative features. One application of this result is type checking for the object-oriented language O’SMALL.

keywords: polymorphic type inference, assignments, recursive types, row variables, subtyping, object-oriented programming

1 Introduction

Types are essential for the ordered evolution of large software systems [3]. This holds for all programming language styles, be they imperative, functional, logical, or object-oriented. Type inference helps to avoid writing redundant information.

The type inference for the $\lambda$-calculus, developed by Hindley and Milner [16, 19], has been extended in various ways in the past. An extension concerned with imperative features was done for the impure functional programming language ML [20, 21]. The resulting type checking is surprisingly restrictive when it comes to assignments. Namely, it is impossible to define a reference to a polymorphic function. Another extension has to do with subtypes. There are many different approaches [1, 2, 8, 18, 24, 9]. Mixing a general subtyping notion and imperative features can be problematic [2]. This may have
been one of the motives for Rémy [23] and Wand [28] to use extensible record types rather than subtyping.

In this paper we will see that indeed we are able to have imperative features and extensible record types, however some limitations suggest that extensible record types are not a complete ersatz for subtypes. What we gain from our solution is complete\(^1\) type inference for the language REC. REC can be the core of an object-oriented programming language with state. An example for such a language is O’SMALL [15]. With the translation from O’SMALL to REC, it is possible to see the constructs of object-oriented programming as mere syntactic sugar. The translation uses wrapper semantics [5] and is described in section 6. For the semantics of REC we rely on the reader’s intuition. We introduce O’SMALL by example (section 2) and list its properties relevant to type checking. Sections 4 and 5 show how to extend the type inference system of Wand by imperative constructs, and section 7 discusses what we have achieved in terms of type inference for O’SMALL.

2 O’small by example

We introduce the object-oriented programming language O’SMALL by examples. The O’SMALL program in figure 1 is about points and circles with Cartesian coordinates in the plane. Points and circles

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

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

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

Figure 1: O’SMALL program with points and circles

\(^1\)‘complete’ means that types do not appear in programs.
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 otherwise invisible 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 the message `m` with argument `a` to the receiver `p`, e.g. `c.move(3,3)` moves a circle object `c` by a certain amount. 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. We can omit the empty parentheses when calling a method without parameters.

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*. Thus, we can still retrieve what is just being redefined. The inherited function *closerToOrg* has not been redefined in the class *Circle*, and it does not have to be redefined in order to be consistent. 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. This is called *late binding*.

The types of the point object `p` and the circle object `c` of figure 1 are as follows. We can imagine the occurring variables as being quantified for each type.

```plaintext
p : [closerToOrg : [distFromOrg : num, ℝ] → bool
  distFromOrg : num
  move : num → num → unit
  x : num
  y : num]

c : [closerToOrg : [distFromOrg : num, ℝ] → bool
  distFromOrg : num
  move : num → num → unit
  r : num
  setR : num → unit
  x : num
  y : num]
```

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 3)) and stands for the infinite set of labels that are not mentioned explicitly. The absence of an ellipsis means, the record type is not extensible. The type unit corresponds to the domain with one element. *unit* is the result type of assignment expressions. If a method ends with an assignment it has a type ending with unit. As type checking is performed on *OBC*, the types of methods appear in a curried version, although parameters are in tuples in *O'Small*-programs. The type of `x` is in fact `unit → num`, but, as the parentheses can be omitted when calling this method, we also omit the type on the left hand side of the arrow.

Some properties of *O'Small* are important for type checking:

- *state*: Objects have assignable instance variables, visible only in the declaring class (*encapsulated instance variables*). A less common feature is that every variable has to be initialized.
  This contrasts to many other languages where uninitialized variables have the value *nil*. 

• **classes**: Classes are no first-class objects, i.e. they cannot be arguments to functions etc.

• **inheritance**: O'SMAL has single inheritance à la Smalltalk [11] using pseudo variables self and super. An extension to inheritance with explicit wrappers [14] permitting the modeling of certain cases of multiple inheritance is possible.

• **parameter passing**: Message parameters are passed and returned by reference. In the body of a method, a formal parameter must not occur on the left hand side of an assignment. This is like "Small" [12], the ancestor of O'SMD.

• **recursion**: There is no direct recursion in declarations. Recursion (including mutual recursion) is achieved by fixed point construction, i.e. via sending messages to self.

### 3 Purely functional type inference

Wand [28] considers type inference for a λ-calculus with records. He extends the approach of Rény [23] by record concatenation and unbounded label sets. Thus, a reduction of the type checking of an object-oriented language without state to Milner's classical type checking [19] is achieved.

The type of a record is the record of the types of its components: or, in other words, the type of a record is a finite function \( L \rightarrow (Type + absent) \) where \( L \) is an infinite set of labels. This does not yield an algebraic signature because \( L \) is infinite. Rény [23] turned this into an algebraic signature and Wand [28] made the extension to infinite label sets, so that we get constructors with the following kinds:

\[
\begin{align*}
\rightarrow & : Type \times Type \Rightarrow Type \\
\mathbin{\prod} & : Field^n \times Extension \Rightarrow Type \\
\text{absent} & : Field \\
\text{pres} & : Type \Rightarrow Field \\
\text{empty} & : Extension
\end{align*}
\]

\( n \) is the number of labels actually appearing in the program under consideration. Therefore, the constructor \( \mathbin{\prod} \) is finite. Unbounded label sets are represented finitely by a finite number of explicit labels and an extension. An empty extension stands for infinitely many labels with absent fields; 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. Record constants have principal types:

\[
\begin{align*}
[] & : \mathbin{\prod}[\mathbin{\prod}_{\text{absent}}] \\
...a & : \mathbin{\prod}_{\text{absent}} (f_1, \ldots, \text{pres}(t), \ldots, f_n, \mathcal{R}) \rightarrow t \\
...a & : \mathbin{\prod}_{\text{pres}} (f_1, \ldots, f_n, \mathcal{R}) \rightarrow t \rightarrow \mathbin{\prod}_{\text{absent}} (f_1, \ldots, \text{pres}(t), \ldots, f_n, \mathcal{R})
\end{align*}
\]

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

\(^2\) this is meant informally and does not imply that types are values.
4 Principal types

O’SMALL does not have classes as first-class values. The full generality of Wand’s type inference system, resulting in the nonexistence of principal types, is not necessary. The absence of principal types results from expressions containing the record concatenation operator \( \oplus \). The concatenation operator appears in the translation function (section 6), but — in the case of single inheritance — it can be eliminated and replaced by fieldwise extension using the constant \( \_ \) with \( a = \_ \). Now we are back to record constants and they have principal types according to section 3.

5 Type inference rules for REC

This is the syntax of REC, the language for which we perform type inference. The variables range over the following syntactic domains: E expressions, M method names (record labels), L identifiers bound by \( \lambda \) or let, D identifiers bound by def, R records.

\[
\begin{align*}
E & ::= \; L \mid D \mid R \mid E_1 E_2 \mid \text{if } E_1 \text{ then } E_2 \text{ else } E_3 \mid \lambda L.E \mid Y E \mid \\
& \quad \text{let } L = E_1 \text{ in } E_2 \mid \text{def } D = E_1 \text{ in } E_2 \mid D := E \\
R & ::= \; [ ] \mid R \text{ with } M = E \mid R.M
\end{align*}
\]

The semantics of REC is call-by-value, because of imperative features. \( Y \) is the fixed point operator. We assume that \( Y \) and basic values are in the initial type environment. let declares a value and def declares a variable. In this context, a variable is a reference to a value. Referencing and dereferencing
is done implicitly. Thus 'return by reference' and 'pass by reference' of \( O\text{SMALL} \) correspond to passing reference values in \( \text{REC} \).

For type inference rules, we follow the notation used in [23]. Let the set of types be denoted by \( T \); 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 \( C = \{ \to, \cdot, [], \text{absent}, \text{pres}, \text{empty} \} \) 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 \( _\varphi \) if they are generic. Types are denoted by letters \( \sigma, \tau \); they have a subscript \( _\varphi \) if generic types are admitted, and a subscript \( _\text{nm} \) if all their type variables are not generalizable. Non-generalizable type variables are needed for occurrences of imperative variables. A type is generic if it may contain generic variables.

**Definition 5.1** A graft is a mapping from \( V \) to \( T \). We use finite grafts only. A graft \( \mu \) is finite if \( \{ \alpha \mid \mu \alpha \neq \alpha \} \) is finite.

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

**Definition 5.2** A generic type \( \tau_\varphi \) is a generic instance of a generic type \( \sigma_\varphi \) iff there exists a graft \( \mu \) such that \( \tau_\varphi = \mu \sigma_\varphi \).

Grafts [17] operate on rational trees as substitutions operate on terms.

**Definition 5.3** An imperative table \( \Sigma \) is a finite set of types (the possible types). A type \( \sigma_m \) is a common instance of an imperative table \( \Sigma = \{ \sigma_{g_1}, \ldots, \sigma_{g_n} \} \) (in symbols: \( \Sigma \upharpoonright \sigma_m \)) iff

\[
\forall 1 \leq i \leq n \exists \mu_i: \mu_i \sigma_{g_i} = \sigma_m
\]

The type inference rules of figure 2 are applied to triples \( (A, E, \tau_\varphi) \) denoted by \( A \vdash E : \tau_\varphi \). \( A \) is the environment of assumptions of the form \( L : \sigma_\varphi \) or \( D : \Sigma \). \( A_x \) denotes all assumptions in \( A \) except those for identifier \( x \). \( E \) is an expression. We note

- \([A, \sigma] \) the generalization of \( \sigma \) with the assumption \( A \). All generalizable variables of \( \sigma \) that do not occur in \( A \) are grafted by new generic variables.
- \([\sigma_\varphi] \) the instantiation of \( \sigma_\varphi \). All generic variables of \( \sigma_\varphi \) are grafted by new non-generic variables.

The inference is split into two parts. The type inference for records (R) is described in section 3, and the type inference for the rest of the language in figure 2. How do the two parts fit together? We can regard record expressions as constants already contained in an initial environment \( A \), and thus as “defined by let”. Therefore the rule \((\text{TAU})\) applies to records. We assume that some further predefined functions are already contained in the assumptions \( A \). The fixed-point operator is among them, and has type \( (\tau \to \tau) \to \tau \).

Readers familiar with type inference systems of purely functional languages will note the absence of a generalization and an instantiation rule [7]. Instead generalization and instantiation are done precisely when variables are declared or when they are used. This is common practice in systems for imperative languages [27]. In contrast to functional languages where the two formulations lead to the same set of inferable types, there are differences here.

We will focus on the “imperative” inference rules, because the others are more or less standard:
(VAR) When an assignable variable is encountered, its type must be a common instance of the possible types in $\Sigma$. The type variables of the common instance are not generalizable.

(DEF) When an assignable variable $D$ is declared, 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 $D$.

(ASS) When an assignable variable $D$ is assigned a value, we check whether the type of the right hand side is contained in the possible types of $D$.

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

6 The translation function

We have claimed that REC can serve as a basis language for object-oriented programming languages. We give a translation from O'SMALL to REC. Comments and examples for the following basic definitions can be found in [15].

**Definition 6.1** We define a higher-order function $\llbracket \cdot \rrbracket : (\alpha \to \beta \to \gamma) \to (\delta \to \alpha) \to (\delta \to \beta) \to \delta \to \gamma$ that makes of a binary operator $\cdot \cdot : \alpha \to \beta \to \gamma$ its self-distributing version denoted by $\llbracket \cdot \rrbracket : (\delta \to \alpha) \to (\delta \to \beta) \to \delta \to \gamma$ and defined by: $a \llbracket \cdot \rrbracket b = \lambda s. (a \cdot s) \cdot (b \cdot s)$

**Definition 6.2** A *record* is a finite mapping from a set of labels to a set of values. A record is denoted by $\left[ \begin{array}{c} z_1 \mapsto v_1 \\ \vdots \\ z_n \mapsto v_n \end{array} \right]$ with labels $z_i$ and values $v_i$. All labels that are not in the list are mapped to $\bot$. The empty record, where all labels are mapped to $\bot$, is denoted by $\emptyset$. Selection of a component $x$ in record $r$ is denoted by $r.x$.

**Definition 6.3** Let $dom(r) = \{ x | r(x) \neq \bot \}$. The *left-preferential concatenation of records* is defined by:

$$ (r \oplus s)(x) = \begin{cases} r(x) & \text{if } x \in dom(r) \\ s(x) & \text{if } x \in dom(s) - dom(r) \\ \bot & \text{otherwise} \end{cases} $$

**Definition 6.4** $\llbracket \cdot \rrbracket : (Object \to Object) \to (Object \to Object)$ takes a function on objects and an object, and yields an object. It is defined by: $f \llbracket \cdot \rrbracket b = (f \cdot b) \oplus b$

As a result, the inheritance function$^3$ $\llbracket \triangleright \rrbracket :$ Wrappere $\to (Class \to Class)$ together with a wrapper $w$ is suited for modifying classes, i.e.:

$$( \llbracket \triangleright \rrbracket w ) : Class \to Class$$

$^3$denoted in infix or prefix, and used with a slightly specialised type
The syntax of O'Small may be inferred from the definition of the translation function or it may be found in [15]. The translation function is denoted by angular brackets and has the type \((\langle \rangle : \text{O'Small} \rightarrow \text{Rec})\). Although the concatenation operator \(\oplus\) is used implicitly in the clause for inheritance, we can do without it and replace it by a finite sequence of "with's". The translation of primitivées is not listed here:

\[
\begin{align*}
\langle K_1 \ldots K_n \ C \rangle &= \langle K_1 \rangle \ldots \langle K_n \rangle \ \langle C \rangle \\
\langle \text{class } I_1 \text{ inheritsFrom } I_2 \text{ def } V \text{ in } M \rangle &= (\lambda \text{self}. \lambda \text{super}. \langle \text{def } V \text{ in } M \rangle) \ [\supset] \langle I_2 \rangle \\
\langle \text{def } V_1 \ldots V_n \text{ in } M \rangle &= \langle \text{def } V_1 \rangle \ldots \langle \text{def } V_n \rangle \ \langle M \rangle \\
\langle \text{var } I := E \rangle &= \langle I \rangle \ \langle E \rangle \\
\langle M_1 \ldots M_n \rangle &= \langle [ \ ] \rangle \ \langle M_1 \rangle \ldots \langle M_n \rangle \\
\langle \text{meth } I(I_1 \ldots I_n) \ C \rangle &= (\lambda I_1 \ldots \lambda I_n). \langle C \rangle \\
\langle C_1 ; \ldots ; C_n \rangle &= \langle \text{let } \text{newVar} = C_1 \rangle \ \langle C_2 ; \ldots ; C_n \rangle \\
\langle I := E \rangle &= \langle I \rangle \ := \langle E \rangle \\
\langle \text{if } E \text{ then } C_1 \text{ else } C_2 \rangle &= \langle \text{if } E \rangle \ \langle C_1 \rangle \ \langle C_2 \rangle \\
\langle \text{def } V_1 \ldots V_n \text{ in } C \rangle &= \langle \text{def } V_1 \rangle \ldots \langle \text{def } V_n \rangle \ \langle C \rangle \\
\langle \text{new } E \rangle &= \langle Y \rangle \ \langle E \rangle \\
\langle E_1 \ O \ E_2 \rangle &= \langle \langle E_1 \rangle \ \langle O \rangle \ \langle E_2 \rangle \rangle \\
\langle E. I(E_1 \ldots E_n) \rangle &= \langle \langle \langle E \rangle . \langle I \rangle \rangle \ \langle E_1 \rangle \ldots \langle E_n \rangle \rangle
\end{align*}
\]

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

7 Assessment

7.1 Recursive types

Figure 3 is a natural example\(^4\) and demonstrates the need for recursive types. The objects of class \texttt{Pair} together with the relation \texttt{leq} define a preorder. The objects of class \texttt{OrderedPair} together with the relation \texttt{leq} and the equality \texttt{eq} define a partial order, because the equality has been redefined and now \texttt{leq} becomes antisymmetric.\(^5\) The type \(t\) of the object \(p\) is circular and defined by the following equation:

\[
t = [\text{eq : [leq : } t \rightarrow \text{bool}, x : \text{num}, y : \text{num}, \mathcal{R}] \rightarrow \text{bool}]
\]

The recursion in the type of \(t\) stems from the redefinition of the equality in the class \texttt{OrderedPair}. The argument \(e\) of \texttt{self.leq} must understand a message \texttt{leq} where the same \texttt{self} is an argument.

Let us see where the circular type comes from. The type check algorithm, which is not described in this paper, proceeds in the following way when it checks the method \texttt{eq} in the class \texttt{OrderedPair}.

\texttt{self} gets the message \texttt{leq(e)} and thus \(e : \alpha \) and \texttt{self} : \( [\texttt{leq : } \alpha \rightarrow \beta, \mathcal{R}] \). Now \(e\) gets the message \texttt{leq(self)} and thus \(e : [\texttt{leq : } \gamma \rightarrow \delta, \mathcal{S}]\).

\(^4\) a modification of an example in [14]

\(^5\) a modification of an example in [14]
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) (self.x + self.y) <= (p.x + p.y)
  meth eq(p) self.x = p.x and self.y = p.y
end

class OrderedPair inheritsFrom Pair
  meth eq(e) self.leq(e) and e.leq(self)
end
def var p := new OrderedPair
  p.set(7,3)
end

Figure 3: O'SMALL program with recursive types

and self : [leq : [leq : γ -> δ, S] -> β, R]. Also γ must be unified with the type of self. The type of self contains γ already, and we get a circular type.

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 [10]. The type checker accepts abstract classes but rejects the creation of their objects (see also section 7.3).

7.2 Imperative features

We can have references to polymorphic functions like f in figure 4. This is more general than in ML [6, 26].

class Id inheritsFrom Base
  meth id(x) x
end
def var f := new Id
  var g := f
  in f.id(3);
  f.id(true)
end

Figure 4: O'SMALL program with assignments

As the types of occurrences of imperative variables have non-generalizable type variables, the object g cannot be used in the same fashion as f. g can only be applied either to 3 or to true but not to both.
There are limitations to our type system with respect to records. The program of figure 5, which does not produce a run time error, is refused by the type checker. The types of a and b cannot be unified because b has a method m that a does not have. The row variable mechanism, that works for arguments of methods, does not work for objects that are known “in advance”, because these objects have non-extensible record types, and the different possible types are unified in our algorithm. It is not possible to give extensible record types to objects, because then, record selection of absent fields 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. [24] may be valuable in this direction of research.

```
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

Figure 5: O'SMALL program with inextensible types
```

### 7.3 Abstract classes

The recognition of abstract classes is a result of the type of classes (section 6): A class has type Object→Object. The type on the left hand side of the arrow (the expected type) is the type of self reference. The type on the right hand side of the arrow (the provided type) is the type of the methods defined in the class. When a class is declared, the two types may be completely incompatible, and the type checker may still accept the class. When an object of the class is created, a fixed point operation takes place, and the two types are unified. Now, common components must be compatible. The expected type is always extensible, whereas the provided type is always non-extensible. If the provided type has a component that is not in the expected type, no problem occurs. However, if the expected type has a component that is not in the provided type, there is a type clash resulting form the non-extensibility of the provided type. The type checker has thus found an abstract class. If a subclass defines the missing components appropriately, objects of the subclass can be created. Summing up, it may be said that abstract classes are recognized “lasily”, i.e. the creation of instances of abstract classes is refused.

In figure 6 class A is abstract. A has a subclass B that is not abstract. B has an instance b. The inferred types are as follows:
6.4: An abstract class

8 Conclusion

We have presented type inference rules for a λ-calculus with records and imperative features. By eliminating the concatenation operator from the language of Wand we got principal types for the purely functional language. Whether the principal-type property extends to imperative features remains to be shown. We have implemented a prototypical type checking algorithm in ML [20, 13] using ML-Yacc [25]. This algorithm realizes the type checking of references to polymorphic functions by delayed unification and by restricting assignments to the declarative scope of variables. It is not contained in this paper, as it is not clear if principal types exist.

The types of objects of subclasses are not in a subtype relation to the types of objects of their superclasses [4]. The test whether a class is abstract comes as a gift with our type inferences.

Our type system is best compared to Palsberg's and Schwartzbach's type inferences [22], because their example language is almost identical to O'Smalls. Their type inferences are based on an entirely different technique, using subtyping and fixed-point derivation rather than unification. A common point is the absence of flow analysis. Their system is more flexible and can check programs that we have to refuse, but it is not incremental. That means that the addition of a single class to a system entails the reconsideration of the whole system. In our approach each class has to be checked once, at the most. Our system is thus especially useful for incremental work, as is the case in rapid prototyping.

Acknowledgements I am grateful to Reinhold Heckmann, Fritz Müller, Joachim Philippi, Georg Sander, Gert Smolka, and Reinhard Wilhelm for comments on drafts of this paper. Thanks to Frank Pfenning for helpful hints concerning the inference rules.
References


