Type inference for polymorphic methods in Java-like languages*

Davide Ancona and Giovanni Lagorio and Elena Zucca

DISI, Univ. of Genova, v. Dodecaneso 35, 16146 Genova, Italy
email: {davide,lagorio,zucca}@disi.unige.it

In languages like C++, Java and C#, typechecking algorithms require methods to be annotated with their parameter and result types, which are either fixed or constrained by a bound.

We show that, surprisingly enough, it is possible to infer the polymorphic type of a method where parameter and result types are left unspecified, as happens in most functional languages. These types intuitively capture the (less restrictive) requirements on arguments needed to safely apply the method.

We formalize our ideas on a minimal Java subset, for which we define a type system with polymorphic types and prove its soundness. We then describe an algorithm for type inference and prove its soundness and completeness. A prototype implementing inference of polymorphic types is available.

1. Introduction

Type inference is the process of automatically determining the types of expressions in a program. That is, when type inference is employed, programmers can avoid writing some (or all) type declarations in their programs.

At the source code level, the situation appears very similar to using untyped (or dynamically typed) languages, as in both cases programmers are not required to write type declarations. However, the similarities end there: when type inference is used, types are statically found and checked by the compiler so no "message not understood" errors can ever appear at runtime (as it may happen when using dynamically typed languages).

To most people the idea of type inference is so tightly tied to functional languages that hearing about one of them automatically springs to mind the other. While it is conceivable to have one without the other, it is a fact that all successful functional languages (like ML, CaML and Haskell) exploit

*This work has been partially supported by MIUR EOS DUE - Extensible Object Systems for Dynamic and Umpredictable Environments.

type inference. Type inference often goes hand in hand with another appealing concept: polymorphism. Indeed, even though type inference and polymorphism are independent concepts, in inferring a type for, say, a function f, it comes quite naturally trying to express "the best" type for f. Indeed, all above mentioned functional languages support both type inference and polymorphism. Outside the world of functional languages, most works on inferring type constraints for object-oriented languages[6,7,14–16] have dealt with structural types. However, in mainstream class-based object-oriented languages with nominal types, typechecking algorithms require methods to be annotated with their parameter types, which are either fixed or constrained by a (nominal) bound.

We show that, surprisingly enough, the approach of inferring the most general function types works smoothly for Java-like languages too. That is, we can define polymorphic types for methods and automatically infer these types when type annotations are omitted. These polymorphic types intuitively express the (minimal) requirements on arguments needed to safely apply the method.

The rest of the paper is organized as follows. In Section 2 we formally define a type system with polymorphic method types for Featherweight Java,[8] in Section 3 we illustrate an algorithm for inferring polymorphic method types, and, finally, in Section 4 we discuss related and further work.

A preliminary version of the ideas exploited in this paper is in a previous work[10] by two of the authors (see Section 4 for a comparison). A small prototype that we have developed can be tried out using any Java-enabled web browser[†].

2. A type system with polymorphic method types

We formalize our approach on a minimal language, whose syntax is given in Figure 1. This language is basically Featherweight Java,[8] a tiny Java subset which has become a standard example to illustrate extensions and new technologies for Java-like languages. However, to focus on the key technical issues and give a compact soundness proof, we do not even consider fields, constructors, and casts, since these features do not pose substantial new problems to our aim[‡]. The only new feature we introduce is the fact that type annotations for parameters can be, besides class names, *type variables* α (in the concrete syntax the user just omits these types and fresh variables

[†]Available at `http://www.disi.unige.it/person/LagorioG/justII/`.

[‡]They can be easily handled by considering new kinds of constraints, see the following.

$$
\begin{array}{rl}
\mathsf{P} & ::= \mathsf{cd}_1 \ldots \mathsf{cd}_n \\
\mathsf{cd} & ::= \texttt{class C extends C}' \ \{\ \mathsf{mds}\ \} \quad (\texttt{C} \neq \texttt{Object}) \\
\mathsf{mds} & ::= \mathsf{md}_1 \ldots \mathsf{md}_n \\
\mathsf{md} & ::= \mathsf{mh}\ \{\texttt{return e;}\} \\
\mathsf{mh} & ::= [\texttt{C}]\ \mathsf{m}(\mathsf{t}_1\ \mathsf{x}_1, \ldots, \mathsf{t}_n\ \mathsf{x}_n) \\
\mathsf{t} & ::= \texttt{C} \mid \alpha \\
\mathsf{e} & ::= \texttt{new C}() \mid \mathsf{x} \mid \mathsf{e}_0.\mathsf{m}(\mathsf{e}_1, \ldots, \mathsf{e}_n)
\end{array}
$$

where class names declared in P, method names declared in mds, and parameter names declared in mh are required to be distinct

Fig. 1. Syntax

are generated by the compiler). Correspondingly, the result type can be omitted, as indicated by the notation [C].

We informally illustrate the approach on a simple example.

```
class A { A m(A anA) { return anA ; }}
class B { B m(B  aB) { return  aB ; }}
class Example {
          polyM(x,y) { return x.m(y) ;                        }
   Object okA()      { return this.polyM(new A(), new A()) ; }
   Object okB()      { return this.polyM(new B(), new B()) ; }
   Object notOk()    { return this.polyM(new A(), new B()) ; }}
```

Polymorphic methods can be safely applied to arguments of different types; however, their possible argument types are determined by a set of constraints, rather than by a single subtyping constraint as in Java generic methods. Intuitively, the polymorphic type of `polyM` should express that the method can be safely applied to arguments of any pair (α, β) s.t. α has a method `m` applicable to β, and the result type is that of `m`. Formally, `polyM` has the polymorphic type $\mu(\delta\ \alpha.\mathtt{m}(\beta)) \Rightarrow \alpha\ \beta \rightarrow \delta$, which means that `polyM` has two parameters of type, respectively, α and β, and returns a value of type δ (right-hand side of $\Rightarrow$), providing the constraint $\mu(\delta\ \alpha.\mathtt{m}(\beta))$ is satisfied (left-hand side of $\Rightarrow$). This happens whenever class α provides a method `m` which can be safely applied to an argument of type β and returns a value of type δ.

According to the type of `polyM`, typechecking of methods `Example.okA` and `Example.okB` should succeed, while typechecking of `Example.notOk` should fail because it invokes `polyM` with arguments of types `A` and `B`, so, in turn, `polyM` requires a method `m` in `A` which can receive a `B` (and there is no such method in the example).

Type environments Δ are sequences of *class signatures* cs, which are triples consisting of a class name $\mathtt{C}$, the name of the parent (that is, the direct superclass) $\mathtt{C}'$ and a sequence of *method signatures* mss. A method signature ms is a tuple consisting of a set of *constraints* Γ, a result type $\mathtt{t}$, a method name m, and sequence of parameter types $\mathtt{t}_1 \ldots \mathtt{t}_n$.

In the simple language we consider, there are only two forms of constraints: the standard subtyping constraint $\mathtt{t} \leq \mathtt{t}'$, and $\mu(\mathtt{t}\ \mathtt{t}_0.\mathsf{m}(\mathtt{t}_1 \ldots \mathtt{t}_n))$, meaning that type $\mathtt{t}_0$ must have a method named m[§] which is applicable to arguments of types $\mathtt{t}_1 \ldots \mathtt{t}_n$ and returns a result of type $\mathtt{t}$. Fields, constructors and casts can be easily handled, as done in another work,[2] by adding other form of constraints.

Clearly, a method type cannot be trivially extracted from the code as happens in standard Java, but a non-trivial inference process is required (see next section). Here we define the type system of the language (Figure 2) which checks that method types are consistent and that the program conforms to them, without specifying how method types can be inferred in practice.

$$(\mathsf{P})\ \frac{\Delta \vdash \mathsf{cd}_i \diamond\ \ \forall i \in 1..n \qquad \vdash \Delta\diamond}{\Delta \vdash \mathsf{cd}_1 \ldots \mathsf{cd}_n\diamond}\ \Delta = \mathsf{cs}_1 \ldots \mathsf{cs}_n$$

$$(\mathsf{cd})\ \frac{\Delta; \mathtt{C} \vdash \mathsf{md}_i \diamond\ \ \forall i \in 1..n}{\Delta \vdash \mathtt{class\ C\ extends\ C'}\ \{\mathsf{md}_1 \ldots \mathsf{md}_n\}\diamond}\ (\mathtt{C}, \mathtt{C}', \mathsf{ms}_1 \ldots \mathsf{ms}_n) \in \Delta$$

$$(\mathsf{md})\ \frac{\begin{array}{l}\Delta; \mathtt{x}_1 : \mathtt{t}_1, \ldots, \mathtt{x}_n : \mathtt{t}_n, \mathtt{this}{:}\mathtt{C}; \Gamma \vdash \mathsf{e} : \mathtt{t}' \\ \Delta; \Gamma \vdash \mathtt{t}' \leq \mathtt{t}\end{array}}{\Delta; \mathtt{C} \vdash [\mathtt{C}_0]\ \mathsf{m}(\mathtt{t}_1\ \mathtt{x}_1, \ldots, \mathtt{t}_n\ \mathtt{x}_n)\ \{\mathtt{return\ e;}\}\diamond}\ \begin{array}{l}\mathit{mtype}(\Delta, \mathtt{C}, \mathsf{m}) = \\ \quad \Gamma \Rightarrow \mathtt{t}_1 \ldots \mathtt{t}_n \rightarrow \mathtt{t} \\ [\mathtt{t} = \mathtt{C}_0]\end{array}$$

$$(\mathsf{call})\ \frac{\Delta; \Pi; \Gamma \vdash \mathsf{e}_i : \mathtt{t}_i\ \forall i \in 0..n \qquad \Delta; \Gamma \vdash \mu(\mathtt{t}\ \mathtt{t}_0.\mathsf{m}(\mathtt{t}_1 \ldots \mathtt{t}_n))}{\Delta; \Pi; \Gamma \vdash \mathsf{e}_0.\mathsf{m}(\mathsf{e}_1, \ldots, \mathsf{e}_n) : \mathtt{t}}$$

$$(\mathsf{new})\ \frac{\Delta; \Gamma \vdash \mathtt{C} \leq \mathtt{C}}{\Delta; \Pi; \Gamma \vdash \mathtt{new\ C}() : \mathtt{C}} \qquad (\mathsf{x})\ \frac{}{\Delta; \Pi; \Gamma \vdash \mathsf{x} : \mathtt{t}}\ \Pi(\mathsf{x}) = \mathtt{t}$$

Fig. 2. Rules for typechecking

[§]This method can be either directly declared or inherited.

By rule (P), a program is well-typed in the type environment Δ if Δ is well-formed (see the comments below), and every class declaration is well-typed in the type environment Δ.

A class declaration is well-formed in Δ (rule (cd)), if all its method declarations are well-formed in Δ and C (needed to correctly type this). The side condition ensures that the class extends the same class and declare the same number of methods as asserted in Δ.

Rule (md) checks that a method declaration is well-typed in Δ and C. Actually, it is a schema that can be instantiated in two different ways: if the return type is not declared (that is, $[\mathtt{C}_0]$ is empty), then the corresponding side conditions $t = \mathtt{C}_0$ must be removed (that is, $[t = \mathtt{C}_0]$ must be empty as well).

The notation $mtype(\Delta, \mathtt{C}, \mathtt{m})$ denotes the type of method m of class C as specified in Δ. The body e must be well-typed in Δ and Π, the local environment assigning the proper types to the parameters and to this; furthermore, e is typechecked assuming that the type constraints in Γ hold. Finally, in Δ it should be derivable from Γ (see the comments below) that the type of e is a subtype of the return type declared for the method.

The last three rules define the typing judgment for expressions, which has form $\Delta; \Pi; \Gamma \vdash \mathtt{e} : \mathtt{t}$, and checks that expression e has type t in the class environment Δ, in the local environment Π, assuming that the constraints in Γ hold.

Rule (call) checks that the expressions denoting the receiver and the arguments are well-typed; furthermore, in Δ the constraint $\mu(\mathtt{t}\ \mathtt{t}_0.\mathtt{m}(\mathtt{t}_1 \ldots \mathtt{t}_n))$ must be derivable from Γ, to ensure that the static type $\mathtt{t}_0$ of the receiver has a method m compatible with the static types $\mathtt{t}_1 \ldots \mathtt{t}_n$ of the arguments.

Rule (new) is standard, except for the constraint $\mathtt{C} \leq \mathtt{C}$, which ensures that a definition for C is available.

For space limitations we have omitted the formal definition of the judgments $\vdash \Delta\diamond$ and $\Delta; \Gamma \vdash \gamma$ (where γ denotes a single constraint) which can be found in an extended version[3] of this paper.

A type environment is well-formed only if it satisfies a number of conditions, including those inherited from FJ: names of declared classes and methods are, respectively, unique in a program and a class declaration, all used class names are declared, and there are no cycles in the inheritance hierarchy. Furthermore, type variables appearing in a list of parameter types must be distinct, and constraints in method types must be *consistent*. Consistency of set of constraints is checked by a normalization procedure

described in the next section. If the procedure succeeds, then the set of constraints is consistent and is transformed into an equivalent but simplified set where constraints are of the form $\alpha \leq \mathtt{t}$ or $\mu(\mathtt{t}\ \alpha.\mathsf{m}(\mathtt{t}_1 \dots \mathtt{t}_n))$. Finally, a type environment is well-formed if overriding of methods is *safe*. The following rule defines the overriding judgment.

$$\text{(overriding)}\ \frac{\begin{array}{l}\Delta;\Gamma\vdash\sigma(\Gamma')\\ \{\Delta;\Gamma\vdash\mathtt{t}_i\leq\mathtt{C}_i \mid \mathtt{t}'_i=\mathtt{C}_i\}\\ \Delta;\Gamma\vdash\sigma(\mathtt{t}')\leq\mathtt{t}\end{array}}{\Delta\vdash \mathsf{mt}\leftarrow\mathsf{mt}'}\quad \begin{array}{l}\mathsf{mt}=\Gamma\Rightarrow\mathtt{t}_1\dots\mathtt{t}_n\rightarrow\mathtt{t},\\ \mathsf{mt}'=\Gamma'\Rightarrow\mathtt{t}'_1\dots\mathtt{t}'_n\rightarrow\mathtt{t}'\\ \mathtt{t}'_i=\alpha_i \implies \sigma(\alpha_i)=\mathtt{t}_i\end{array}$$

This rule states that a method type safely overrides another if the constraints in the heir can be derived from those of its parent, modulo a substitution that maps type variables used as parameter types in the heir into the corresponding parameter types in the parent. This condition intuitively guarantees that the method body of the heir (which has been typechecked under the heir constraints) can be safely executed under its parent constraints. Moreover, parameter types in the heir which are classes must be more generic, and return type more specific. Note that on monomorphic methods the definition reduces to contravariance for parameter types and covariance for return type, hence to a more liberal condition than in standard FJ and Java.

The *entailment* judgment $\Delta;\Gamma\vdash\gamma$ is valid if in Δ the constraint γ is entailed by Γ. We will also write $\Delta\vdash\gamma$ for $\Delta;\emptyset\vdash\gamma$ (this means that γ hold in Δ).

The rules are available in the extended version[3] and are pretty straightforward, except that for constraints of the form $\mu(\mathtt{t}\ \mathtt{C}_0.\mathsf{m}(\mathtt{t}_1\dots\mathtt{t}_n))$:

$$(\mu)\ \frac{\begin{array}{l}\{\Delta;\Gamma\vdash\mathtt{t}_i\leq\mathtt{C}_i \mid \mathtt{t}'_i=\mathtt{C}_i\}\\ \Delta;\Gamma,\mu(\mathtt{t}\ \mathtt{C}_0.\mathsf{m}(\mathtt{t}_1\dots\mathtt{t}_n))\vdash\sigma(\Gamma')\end{array}}{\Delta;\Gamma\vdash\mu(\mathtt{t}\ \mathtt{C}_0.\mathsf{m}(\mathtt{t}_1\dots\mathtt{t}_n))}\quad \begin{array}{l}\mathit{tvars}(\sigma(\Gamma'))\subseteq\\ \qquad \mathit{tvars}(\mu(\mathtt{t}\ \mathtt{C}_0.\mathsf{m}(\mathtt{t}_1\dots\mathtt{t}_n)))\\ \mathit{mtype}(\Delta,\mathtt{C}_0,\mathsf{m})=\Gamma'\Rightarrow\mathtt{t}'_1\dots\mathtt{t}'_n\rightarrow\mathtt{t}'\\ \mathtt{t}'_i=\alpha_i \implies \sigma(\alpha_i)=\mathtt{t}_i\\ \sigma(\mathtt{t}')=\mathtt{t}\end{array}$$

Here the substitution σ ensures that the types of the arguments and of the returned value of the constraint matches the corresponding declaration of the method in Δ. The side condition $\mathit{tvars}(\sigma(\Gamma'))\subseteq\mathit{tvars}(\mu(\mathtt{t}\ \mathtt{C}_0.\mathsf{m}(\mathtt{t}_1\dots\mathtt{t}_n)))$ ensures that the type variables of $\sigma(\Gamma')$ are included in those of $\mu(\mathtt{t}\ \mathtt{C}_0.\mathsf{m}(\mathtt{t}_1\dots\mathtt{t}_n))$, in order to avoid unwanted clashes with the variables in Γ. This condition can be always satisfied either by a

proper α-renaming of variables, or by substituting with ground terms the variables in Γ' which do not appear in $\mathtt{t}'_1 \ldots \mathtt{t}'_n \to \mathtt{t}'$.

Let us consider an instantiation of the rule above in the typechecking of the invocation `this.polyM(new A(), new A())` in method `Object okA()` in our initial code example; such an invocation typechecks since the judgment $\Delta \vdash \mu(\mathtt{A\ Example.polyM(A\ A)})$ is valid, with Δ the type environment corresponding to the program. Indeed, $\mathit{mtype}(\Delta, \mathtt{Example}, \mathtt{polyM}) = \mu(\gamma\ \alpha.\mathtt{m}(\beta)) \Rightarrow \alpha\ \beta \to \gamma$ and, by substituting α, β and γ with `A` we get $\mu(\mathtt{A\ A.m(A)})$ which holds in Δ.

Note that in the premise of the rule we add $\mu(\mathtt{t\ C}_0.\mathtt{m}(\mathtt{t}_1 \ldots \mathtt{t}_n))$ to Γ. This is necessary to avoid infinite proof trees when typechecking recursive methods, as in the following example:

```
class C {
  m (x) { return x.m(x);}
  Object test () { return this.m(this); }
}
```

Here, polymorphic method `m` has type $\mu(\beta\ \alpha.\mathtt{m}(\alpha)) \Rightarrow \alpha \to \beta$. The invocation `this.m(this)` in method `test` typechecks since the judgment $\Delta \vdash \mu(\mathtt{C\ C.m(C)})$ holds, with Δ the type environment corresponding to the program. Indeed, $\mathit{mtype}(\Delta, \mathtt{C}, \mathtt{m}) = \mu(\beta\ \alpha.\mathtt{m}(\alpha)) \Rightarrow \alpha \to \beta$ and, by substituting α and β with `C` we get the constraint $\mu(\mathtt{C\ C.m(C)})$ which do not need to be proved again.

The type system with polymorphic method types we have defined is sound, that is, expressions which can be typed by using (the type information corresponding to) a well-formed program `P` can be safely executed w.r.t. this program, where reduction rules for $\to_{\mathtt{P}}$ are standard. For lack of space we omit them here, but they can be found in the extended version[3] of this paper. This means in particular that these expressions are ground and do not require type constraints. The proof is given by the standard subject reduction and progress properties. The proof schema is similar to that given for Featherweight GJ;[9] roughly, in Featherweight GJ only a kind of constraints on type variables is considered, that is, that they satisfy their (recursive) upper bound. The details of these proofs can be found in the aforementioned technical report.

Theorem 2.1 (Progress). *If* $\Delta \vdash P\diamond$ *and* $\Delta; \emptyset; \emptyset \vdash \mathtt{e} : t$, *then either* $\mathtt{e} = \mathtt{new}\ C()$ *or* $\mathtt{e} \to_P \mathtt{e}'$ *for some* $\mathtt{e}'$.

Theorem 2.2 (Subject reduction). *If* $\Delta \vdash P\diamond$ *and* $\Delta; \Pi; \emptyset \vdash \mathtt{e} : t$, $\mathtt{e} \to_P \mathtt{e}'$, *then* $\Delta; \Pi; \emptyset \vdash \mathtt{e} : t'$, $\Delta \vdash t' \leq t$.

3. Inferring polymorphic method types

In this section we show how the typechecking defined in Section 2 can be made effective by defining an algorithm for generating method types and another for checking consistency, by normalization, of type constraints of method types.

The first algorithm can be derived in a straightforward way by a set of rules[3] which have been omitted here for space limitations. Rules are defined by following an approach similar to that adopted[2] for achieving principality in Java: each rule just records all constraints necessary to successfully typecheck a certain kind of expression, without performing any check; hence, generation of method types always succeeds.

For instance, the rule for generating constraints for a method invocation is as follows:

$$\text{(call)}\frac{\Pi \vdash \mathsf{e}_i : \Gamma_i \Rightarrow \mathtt{t}_i \ \forall i \in 0..n}{\Pi \vdash \mathsf{e}_0.\mathsf{m}(\mathsf{e}_1,\ldots,\mathsf{e}_n) : \Gamma_0,\ldots,\Gamma_n, \mu(\alpha\ \mathtt{t}_0.\mathsf{m}(\mathtt{t}_1\ldots\mathtt{t}_n)) \Rightarrow \alpha}\ \alpha \text{ fresh}$$

The judgment for constraint generation has form $\Pi \vdash \mathsf{e} : \Gamma \Rightarrow \mathtt{t}$ where the local variable environment Π and the expression e are the input values, whereas the set of constraints Γ and the type $\mathtt{t}$ are the output values of the generation process. Note that the type $\mathtt{t}$ of the expression is only needed for generating type constraints and is not used for performing a real typechecking.

The algorithm that normalizes a set of constraints is described in pseudocode in Figure 3, together with its pre- and postcondition. If normalization fails, then the corresponding set of constraints is not consistent. The variable all contains the current set of constraints, and the variable done keeps track of those which have already been checked. Termination of the process is guaranteed by the use of done and by the fact that, given a certain program, the set of all possible constraints which can be generated from all is finite.

We write $\Delta \vdash \Gamma \sim \Gamma'$ to denote that $\Delta;\Gamma \vdash \Gamma'$ and $\Delta;\Gamma' \vdash \Gamma$ hold.

Theorem 3.1 (Correctness of the algorithm). *The algorithm in Figure 3 is correct w.r.t. the given pre- and postcondition.*

Theorem 3.2 (Soundness of type inference). *If* $\vdash\ \mathsf{cd}_1 \ldots \mathsf{cd}_n\ :\ \Delta$, *then* $\Delta \vdash \mathsf{cd}_1 \ldots \mathsf{cd}_n \diamond$.

Theorem 3.3 (Completeness of type inference). *If* $P = \mathsf{cd}_1 \ldots \mathsf{cd}_n$, $\vdash \mathsf{cd}_i : \mathit{cs}_i$ *for all* $i \in 1..n$ *and the simplification algorithm fails on* $\mathit{cs}_1 \ldots \mathit{cs}_n$, *then there exists no* Δ *s.t.* $\Delta \vdash \mathsf{cd}_1 \ldots \mathsf{cd}_n \diamond$.

```
{all==Γ && done==∅ &&!failure}
while (∃γ ∈ (all \ done not in normal form)&&!failure) {
  done = done ∪ {γ};
  switch γ
    case C ≤ C′:
      if (Δ⊬C ≤ C′) failure = true;
    case μ(α C.m(t1 ... tn)):
      mt = mtype(Δ, C, m);
      if (mt undefined) failure = true;
      else
        let mt = Γ′ ⇒ t′1 ... t′m → t′ in
          if (m!=n) failure = true;
          else
            subst_ι = {αi ↦ ti | t′i = αi};
            subst_o = {α ↦ subst_ι(t′)};
            all = subst_o(all) ∪ subst_ι(Γ′ ∪ {ti ≤ Ci | t′i = Ci});
            done = subst(done);
    default:
      failure = true
    }
    {!failure==(∃Γ^nf in normal form and σ s.t. Δ⊢Γ^nf ∼ σ(Γ))};
```

Fig. 3. Simplification of constraints

More details on these results and their proofs can be found in the aforementioned extended technical report.

Extension to full FJ When considering full FJ, the other forms of constraints which come out can be easily accommodated in the schema. For instance, constraints of the form $\phi(\mathtt{t}'\ \mathtt{t.f})$ (type $\mathtt{t}$ must have a field named $\mathtt{f}$ of type $\mathtt{t}'$) and $\mathtt{t} \sim \mathtt{t}'$ ($\mathtt{t}$ must be a subtype of $\mathtt{t}'$ or conversely) can be handled as the $\mathtt{t} \leq \mathtt{t}'$ constraints, in the sense that they must be just checked, whereas constraints of the form $\kappa(\mathtt{t}(\mathtt{t}_1 \ldots \mathtt{t}_n))$, meaning that type $\mathtt{t}$ must have a constructor applicable to arguments of types $\mathtt{t}_1 \ldots \mathtt{t}_n$, are a simpler version of the $\mu(\mathtt{t}'\ \mathtt{t.m}(\mathtt{t}_1 \ldots \mathtt{t}_n))$ constraints, in the sense that they can generate new constraints when checked.

4. Related and further work

As mentioned, the idea of omitting type annotations in method parameters has been preliminarily investigated in a previous work.[10] However, the key problem of solving recursive constraint sets is avoided there by imposing a

rather severe restriction on polymorphic methods.

The type inference algorithm presented here can be seen as a generalization of that for compositional compilation of Java-like languages.[2] Indeed, the idea leading to the work in this paper came out very nicely by realizing that the constraint inference algorithm adopted there for compiling classes in isolation extends smoothly to the case where parameter types are type variables as well.

However, there are two main differences: first, the compositional compilation algorithm[2] only eliminates constraints, whereas here new constraints can be added since other methods can be invoked in a method's body, thus making termination more an issue. Secondly, since we may also have type variables as method parameter types, substitutions are not necessarily ground.

Type inference in object oriented languages has been studied before; in particular, an algorithm for a basic language with inheritance, assignments and late-binding has been described.[13,15] An improved version of the algorithm is called CPA (Cartesian Product Algorithm).[1] In these approaches types are set of classes, like in Strongtalk,[4] a typechecker for Smalltalk. More recently, a modified CPA[16] has been designed which introduces *conditional constraints* and resolves the constraints by least fixed-point derivation rather than unification. Whereas the technical treatment based on constraints is similar to ours, their aim is analyzing standard Java programs (in order to statically verify some properties as downcasts correctness) rather than proposing a polymorphic extension of Java.

As already pointed out, while in Java the only available constraint on type variables is subtyping, in our approach we can take advantage of a richer set of constraints, thus making method types more expressive; furthermore, while our system is based on type inference, in Java the type variables and the constraints associated with a generic method are not inferred, but have to be explicitly provided by the user.

Our type constraints are more reminiscent of *where-clauses*[5,12] used in the *PolyJ* language. In *PolyJ* programmers can write parameterized classes and interfaces where the parameter has to satisfy constraints (the where-clauses) which state the signatures of methods and constructors that objects of the actual parameter type must support. The fact that our type constraints are related to methods rather than classes poses the additional problem of handling recursion. Moreover, our constraints for a method may involve type variables which correspond not only to the parameters, but also to intermediate result types of method calls.

Type inference has been deeply investigated in the context of functional languages since the early 80s, where many systems proposed in literature are based on the Hindley/Milner system[11] with constraints; the relation between our approach and that system deserves further investigation.

We have shown how to infer the polymorphic type of a method where parameter and result types are left unspecified, as it happens in most functional languages. Polymorphic method types are expressed by a set of constraints which intuitively correspond to the minimal requirements on argument types needed to safely apply the method. In this way the type system proposed here turns out to be very flexible.

We have also developed a small prototype that implements the described type inference and simplification of constraints (though the implemented overriding rule is simpler, so less liberal, than the one described here). This prototype, written in Java, can be tried out using any web browser¶.

We believe this is a nice result, which bridges the world of type inference for polymorphic functions and the one of object-oriented languages with nominal types, showing a relation which deserves further investigation.

On the more practical side, our work can serve as basis for developing extensions of Java-like languages which allow developers to forget about (some) type annotations as happens in scripting languages, gaining some flexibility without losing static typing. A different design alternative is to let programmers to specify (some) requirements on arguments.

Finally, the system presented here is not complete w.r.t. the type system defined in Section 2, but we are planning to investigate whether completeness can be achieved.

References

1. Ole Agesen. The cartesian product algorithm. In W. Olthoff, editor, *ECOOP'05 - Object-Oriented Programming*, volume 952 of *Lecture Notes in Computer Science*, pages 2–26. Springer, 1995.
2. Davide Ancona, Ferruccio Damiani, Sophia Drossopoulou, and Elena Zucca. Polymorphic bytecode: Compositional compilation for Java-like languages. In *ACM Symp. on Principles of Programming Languages 2005*. ACM Press, January 2005.
3. Davide Ancona, Giovanni Lagorio, and Elena Zucca. Type inference for polymorphic methods in Java-like languages. Technical report, Dipartimento di Informatica e Scienze dell'Informazione, Università di Genova, 2007. Submitted for journal publication.

¶Available at `http://www.disi.unige.it/person/LagorioG/justII/`.

4. Gilad Bracha and David Griswold. Strongtalk: Typechecking Smalltalk in a production environment. In *ACM Symp. on Object-Oriented Programming: Systems, Languages and Applications 1993*, pages 215–230, 1993.
5. Mark Day, Robert Gruber, Barbara Liskov, and Andrew C. Myers. Subtypes vs. where clauses: Constraining parametric polymorphism. In *ACM Symp. on Object-Oriented Programming: Systems, Languages and Applications 1995*, volume 30(10) of *SIGPLAN Notices*, pages 156–168, 1995.
6. Jonathan Eifrig, Scott F. Smith, and Valery Trifonov. Sound polymorphic type inference for objects. In *ACM Symp. on Object-Oriented Programming: Systems, Languages and Applications 1995*, volume 30(10) of *SIGPLAN Notices*, pages 169–184, 1995.
7. Jonathan Eifrig, Scott F. Smith, and Valery Trifonov. Type inference for recursively constrained types and its application to OOP. In *Mathematical Foundations of Programming Semantics*, volume 1 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 1995.
8. Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *ACM Symp. on Object-Oriented Programming: Systems, Languages and Applications 1999*, pages 132–146, November 1999.
9. Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.
10. Giovanni Lagorio and Elena Zucca. Introducing safe unknown types in Java-like languages. In L.M. Liebrock, editor, *OOPS'06 - Object-Oriented Programming Languages and Systems, Special Track at SAC'06 - 21st ACM Symp. on Applied Computing*, pages 1429–1434. ACM Press, 2006.
11. Robin Milner. A theory of type polymorphism in programming. *Journ. of Computer and System Sciences*, 17(3):348–375, 1978.
12. Andrew C. Myers, Joseph A. Bank, and Barbara Liskov. Parameterized types for Java. In *ACM Symp. on Principles of Programming Languages 1997*, pages 132–145. ACM Press, 1997.
13. J. Palsberg and M. I. Schwartzbach. *Object-Oriented Type Systems*. John Wiley & Sons, 1994.
14. Jens Palsberg. Type inference for objects. *ACM Comput. Surv.*, 28(2):358–359, 1996.
15. Jens Palsberg and Michael I. Schwartzbach. Object-oriented type inference. In *ACM Symp. on Object-Oriented Programming: Systems, Languages and Applications 1991*, pages 146–161, 1991.
16. Tiejun Wang and Scott F. Smith. Precise constraint-based type inference for Java. In *ECOOP'01 - European Conference on Object-Oriented Programming*, volume 2072, pages 99–117. Springer, 2001.