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, programmers can avoid writing some (or all) type declarations in their programs when type inference is employed.

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 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[1–5] 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,[6] illustrate its meaning on some examples and show that it is is sound, in the usual sense that well-typed programs never go stuck. In Section 3 we illustrate an algorithm for inferring polymorphic method types, by first deriving constraints for any method in isolation and then simplifying these constraints by checking that mutual assumptions are satisfied. In Section 4 we briefly describe the prototype we have developed and discuss other implementation issues. Finally, in Section 5 we discuss related work and in Section 6 we summarize our contribution and draw some directions for further research.

A preliminary version of the ideas exploited in this paper is in a previous work[7] by two of the authors (see the Conclusion for a comparison).

2. A type system with polymorphic method types

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

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

We formalize our approach on a minimal language, whose syntax is given in Figure 1. This language is basically Featherweight Java,[6] 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 are automatically 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()) ;}}
```

In this example, method `polyM` is the only polymorphic method, all the others are standard methods. 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, method `polyM` has the polymorphic type $\mu(\gamma\ \alpha.\mathtt{m}(\beta)) \Rightarrow \alpha\ \beta \rightarrow \gamma$, which means that `polyM` has two parameters of type α and β and returns a value of type γ (right-hand side of $\Rightarrow$), providing that the constraint $\mu(\gamma\ \alpha.\mathtt{m}(\beta))$ is satisfied (left-hand side of $\Rightarrow$), that is, class α has a method `m` which can be safely applied to an argument of type β by returning a value of type γ.

In a type environment where we have[‡] this type for `Example.polyM`, type-checking of methods `Example.okA` and `Example.okB` should succeed, while type-checking 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).

We will see later other examples illustrating how chains of method calls and recursion are handled. Type environments Δ are formally defined in Figure 2. They are sequences of *class signatures*, which are triples consisting of a class name, the name of the parent and a sequence of *method signatures*.

[†]They can be easily handled by just considering new kinds of constraints, see the following.
[‡]We will see in Section 3 how to *infer* this type.

$$\begin{aligned}
\Delta &::= \mathsf{cs}_1 \ldots \mathsf{cs}_n \\
\mathsf{cs} &::= (\mathtt{C}, \mathtt{C}', \mathsf{mss}) \\
\mathsf{mss} &::= \mathsf{ms}_1 \ldots \mathsf{ms}_n \\
\mathsf{ms} &::= \Gamma \Rightarrow \mathtt{t}\ \mathsf{m}(\mathtt{t}_1 \ldots \mathtt{t}_n) \\
\Gamma &::= \gamma_1 \ldots \gamma_n \\
\gamma &::= \mathtt{t} \leq \mathtt{t}' \mid \mu(\mathtt{t}\ \mathtt{t}_0.\mathsf{m}(\mathtt{t}_1 \ldots \mathtt{t}_n))
\end{aligned}$$

Fig. 2. Type environments

A method signature is a tuple consisting of a set of *constraints* Γ, a result type, a method name, and sequence of parameter types.

In the simple language we consider, there are only two forms of constraints: $\mathtt{t} \leq \mathtt{t}'$, meaning that type $\mathtt{t}$ must be a subtype of $\mathtt{t}'$, 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 (either directly declared or inherited) method named m applicable to arguments of types $\mathtt{t}_1 \ldots \mathtt{t}_n$ and giving, for these argument types, a result of type $\mathtt{t}$. Fields, constructors and casts can be easily handled, as done in another work,[8] adding constraints of the form: $\phi(\mathtt{t}'\ \mathtt{t}.\mathsf{f})$, meaning that type $\mathtt{t}$ must have a (either directly declared or inherited) field named f of type $\mathtt{t}'$, $\kappa(\mathtt{t}(\mathtt{t}_1 \ldots \mathtt{t}_n))$, meaning that type $\mathtt{t}$ must have a constructor applicable to arguments of types $\mathtt{t}_1 \ldots \mathtt{t}_n$, and $\mathtt{t} \sim \mathtt{t}'$, meaning that either type $\mathtt{t}$ must be a subtype of $\mathtt{t}'$ or conversely.

Note that, w.r.t. the standard Java case, type environments cannot be trivially extracted from (either source or binary) code by just taking method headers, since we also need constraints. Instead, constructing the type environment associated with a program requires a non-trivial inference process, which will be described in the next section. In practice, we expect this process to be applied to some source code, say `S`, generating bytecode `B` enriched by its constraints. In this way, separate compilation can be implemented as it is in standard Java, since source code using this bytecode could be compiled by just extracting in a trivial way the type environment from `B`. Rules for typechecking a program in a given type environment are given in Figure 3.

By rule (P), a program is well-typed in the type environment Δ if Δ is well-formed ($\vdash \Delta\diamond$), and every class declaration conforms to the type environment Δ. The judgment $\vdash \Delta\diamond$ is defined in Figure A2 in the Appendix.

By rule (cd), in the type environment Δ we can derive a class signature from a class declaration with name `C` if in Δ and current class `C` (needed as type of `this`) we can derive for each method declaration the given method signature.

Rules (md-α) and (md-`C`) check that the body e of m conforms to the *method type* $\Gamma \Rightarrow \mathtt{t}_1 \ldots \mathtt{t}_n \rightarrow \mathtt{t}$ found in Δ, and extracted by the function *mtype*, see Figure A1 in the Appendix. More precisely, e is typechecked in Δ, under the method constraints Γ, and in a *parameter environment* Π which assigns to the

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

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

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

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

$$(\mathsf{x})\,\frac{}{\Delta; \Pi; \Gamma \vdash \mathsf{x} : \mathtt{t}}\ \Pi(\mathsf{x}) = \mathtt{t}$$

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

$$(\mathsf{new})\,\frac{\Delta; \Gamma \vdash \mathtt{C} \leq \mathtt{C}}{\Delta; \Pi; \Gamma \vdash \mathtt{new\ C()} : \mathtt{C}}$$

Fig. 3. Rules for typechecking

implicit parameter `this` the current class, and to each parameter the corresponding type. Moreover, if an explicit return type was written by the user, then this type must conform with the return type found in Δ (md-C).

The *entailment* judgment $\Delta; \Gamma \vdash \gamma$ is formally defined in Figure A3 in the following. Intuitively, it holds if the constraint γ either holds in Δ or is one of the constraints in Γ. We will also write $\Delta \vdash \gamma$ for $\Delta; \emptyset \vdash \gamma$.

The last three rules define the typing judgment for expressions, which has form $\Delta; \Pi; \Gamma \vdash \mathsf{e} : \mathtt{t}$. The type environment Δ is needed to perform type checks involving existing classes (for instance, $\mathtt{C}_1 \leq \mathtt{C}_2$), whereas the constraints Γ express requirements on the parameter types which are just assumed to hold for the current method.

Rule (x) is standard.

By rule (call), in the type environment Δ and parameter environment Π, under the constraints Γ, we can typecheck a method call if the receiver and arguments can be successfully typechecked, and the type of the receiver has a method with the given name applicable to the argument types. The type of the method call is the return type of the method for the given argument types.

Rule (new) is standard, except for the constraint $\mathtt{C} \leq \mathtt{C}$, which encodes the fact that `C` must be an existing class in order for the creation expression to be correct (see rule ($\leq$-refl-class) in Figure A3).

The rules for well-formed type environments can be found in Figure A2 in the Appendix.

A type environment is well-formed only if it satisfies a number of conditions, including standard FJ and Java conditions (names of declared classes and methods are unique in a program and class declaration, respectively, all used class names are declared, there are no cycles in the inheritance hierarchy). Moreover, type variables appearing as parameter types must be distinct, and methods must use disjoint sets of variables (this condition prevents variable clashes in rule (μ) in Figure A3). Constraints in method types must be *in normal form*, that is, of the form $\gamma ::= \alpha \leq \mathtt{t} \mid \mu(\mathtt{t}\ \alpha.\mathsf{m}(\mathtt{t}_1 \ldots \mathtt{t}_n))$; intuitively, this means that they correspond to requirements on argument types. Finally, overriding must be *safe* in a sense which goes beyond that of standard Java since we have also to check that constraints in the heirs are, roughly, no stronger than those in their parent, see rule (overriding) in Figure 2.

Figure A3, in the Appendix, contains the formal definition of the entailment judgment. The rules are pretty straightforward, except for (μ),

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

where σ denotes a *substitution* mapping type variables into types. This rule states that a constraint $\mu(\mathtt{t}\ \mathtt{C}_0.\mathsf{m}(\mathtt{t}_1\ldots\mathtt{t}_n))$ holds in a given type environment Δ, under assumptions Γ, if in Δ there exists a method applicable to the given argument types leading to the given return type. Applicability of a method goes beyond that of standard Java, since, for parameter types which are type variables, the method is applicable only if by replacing these variables by the corresponding argument types we obtain a set of provable constraints.

Referring to the previous example, for instance the invocation `this.polyM(new A(), new A())` in method `Object okA()` typechecks since the judgment $\Delta \vdash \mu(\mathtt{A}\ \mathtt{Example.polyM}(\mathtt{A}\ \mathtt{A}))$ holds, with Δ the type environment corresponding to the program. This judgment holds since $\mathit{mtype}(\Delta,\mathtt{Example},\mathtt{polyM}) = \mu(\gamma\ \alpha.\mathtt{m}(\beta)) \Rightarrow \alpha\ \beta \rightarrow \gamma$ and, by substituting α, β and γ with `A` we get $\mu(\mathtt{A}\ \mathtt{A.m}(\mathtt{A}))$ which holds in Δ.

Note that in the premise of the rule we add $\mu(\mathtt{t}\ \mathtt{C}_0.\mathsf{m}(\mathtt{t}_1\ldots\mathtt{t}_n))$ to Γ. This is needed to be able to typecheck recursive methods avoiding infinite proof trees, 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 $\mu(\beta\ \alpha.\mathtt{m}(\alpha)) \Rightarrow \alpha \rightarrow \beta$. The invocation `this.m(this)` in method `test` typechecks since the judgment $\Delta \vdash \mu(\mathtt{C}\ \mathtt{C}.\mathtt{m}(\mathtt{C}))$ holds, with Δ the type environment corresponding to the program. This judgment holds since $\mathit{mtype}(\Delta, \mathtt{C}, \mathtt{m}) = \mu(\beta\ \alpha.\mathtt{m}(\alpha)) \Rightarrow \alpha \rightarrow \beta$ and, by substituting α and β with `C` we get the constraint $\mu(\mathtt{C}\ \mathtt{C}.\mathtt{m}(\mathtt{C}))$ which should not be proved again.

The following rule defines the overriding judgment.

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

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 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 $\rightarrow_{\mathtt{P}}$ are standard and shown in Figure **??** in the Appendix. 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, and requires a number of lemmas (see the Appendix). 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.

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

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

3. Inferring polymorphic method types

The algorithm which computes Γ^{nf} is described in pseudocode in Figure 4, together with its pre- and postcondition. The variable all contains the current set of constraints, and the variable done keeps trace of those which have already been checked. We write $\Delta \vdash \Gamma \sim \Gamma'$ to denote that $\Delta;\Gamma \vdash \Gamma'$ and $\Delta;\Gamma' \vdash \Gamma$ hold. Some examples of this algorithm at work are shown in Section 6.1 in the Appendix.

```
{all==$\Gamma$ && done==$\emptyset$ &&!failure}
while ($\exists\gamma \in$ (all $\setminus$ done not in normal form)&&!failure) {
  done = done $\cup$ {$\gamma$};
  switch $\gamma$
    case $\mathtt{C} \leq \mathtt{C}'$:
      if ($\Delta \not\vdash \mathtt{C} \leq \mathtt{C}'$) failure = true;
    case $\mu(\alpha\ \mathtt{C.m}(\mathtt{t}_1 \ldots \mathtt{t}_n))$:
      mt = $mtype(\Delta, \mathtt{C}, \mathtt{m})$;
      if (mt undefined) failure = true;
      else
        let mt = $\Gamma' \Rightarrow \mathtt{t}'_1 \ldots \mathtt{t}'_m \rightarrow \mathtt{t}'$ in
          if ($m$!=$n$) failure = true;
          else
            subst = $\{\alpha_i \mapsto \mathtt{t}_i \mid \mathtt{t}'_i \equiv \alpha_i\}$;
            subst = subst $\cup \{\alpha \mapsto$ subst($\mathtt{t}'$)$\}$;
            all = subst(all $\cup\ \Gamma' \cup \{\mathtt{t}_i \leq \mathtt{C}_i \mid \mathtt{t}'_i \equiv \mathtt{C}_i\}$);
            done = subst(done);
  }
  {!failure==($\exists\Gamma^{nf}$ in normal form and $\sigma$ s.t. $\Delta \vdash \Gamma^{nf} \sim \sigma(\Gamma)$)};
```

Fig. 4. Simplification of constraints

Note that the switch construct covers all possible cases. Indeed, since constraints in Γ are generated by the type inference algorithm (shown in Figure A4 in the Appendix), and substitution always apply only to parameter types, it is easy to see that we never get constraints of form $\mathtt{C} \leq \alpha$ or $\mu(\mathtt{C}'\ \mathtt{C.m}(\mathtt{t}_1 \ldots \mathtt{t}_n))$; this condition is omitted in the formal invariant for simplicity.

Theorem 3.1 (Correctness of the algorithm). *The algorithm in Figure 4 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 $P = \mathsf{cd}_1 \dots \mathsf{cd}_n$, $\vdash \mathsf{cd}_i : \mathsf{cs}_i$ for all $i \in 1..n$ and the simplification algorithm fails on $\mathsf{cs}_1 \dots \mathsf{cs}_n$, then there exists no Δ s.t. $\Delta \vdash \mathsf{cd}_1 \dots \mathsf{cd}_n \diamond$.*

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 $\phi(\mathtt{t}'\ \mathtt{t}.\mathsf{f})$ (type $\mathtt{t}$ must have a field named 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 $\kappa(\mathtt{t}(\mathtt{t}_1 \dots \mathtt{t}_n))$, meaning that type $\mathtt{t}$ must have a constructor applicable to arguments of types $\mathtt{t}_1 \dots \mathtt{t}_n$, are a simpler version of the $\mu(\mathtt{t}'\ \mathtt{t}.\mathsf{m}(\mathtt{t}_1 \dots \mathtt{t}_n))$ constraints, in the sense that they can generate new constraints when checked.

4. Implementation

We have developed a small prototype that implements the type inference and simplification of constraints described, respectively, in Figure A4 and 4. This prototype, written in Java, can be tried out using any Java-enabled web browser[§].

Currently, it supports only the language described in the paper, and implements an overriding rule which is simpler (and less liberal) than that presented on page 7:

$$\text{(simple-overriding)}\ \frac{\sigma(\Gamma') \subseteq \Gamma \qquad \sigma(\mathtt{t}_i') = \mathtt{t}_i\ \forall\, i \in 0..n}{\Delta \vdash (\Gamma \Rightarrow \mathtt{t}_1 \dots \mathtt{t}_n \rightarrow \mathtt{t}_0) \leftarrow (\Gamma' \Rightarrow \mathtt{t}_1' \dots \mathtt{t}_n' \rightarrow \mathtt{t}_0')}$$

However, we are working on an extension implementing the full overriding rule and supporting other language features as constructors, fields and some statements in order to experiment with more significant examples.

Adding new constructs should not pose particular challenges, since it boils down to adding new kinds of constraints to model features like field accesses, constructor invocations and type equality. All these kinds of constraints are conceptually simpler than the two we already handle.

Future work includes real compilation of sources into standard Java source/bytecode, where the challenging part is the translation of invocations of polymorphic methods, since every JVM (Java Virtual Machine) invocation instruction must be fully annotated with the static *standard* types of target and arguments, and not with type variables. Java generics, being oblivious to the JVM, are of no help.

By using reflection, as suggested by Lagorio and Zucca,[7] invocations involving polymorphic types can be easily translated into standard Java source/bytecode. Producing *efficient* Java bytecode, on the other hand, is more challenging.

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

Reflection could be probably avoided instantiating polymorphic methods into sets of monomorphic ones, a là C++ templates, but this may result in code bloat.

5. Related Work

As mentioned in the Introduction, the idea of omitting type annotations in method parameters has been preliminarily investigated in a previous work.[7] However, there the key problem of solving recursive constraint sets is avoided 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.[8] 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.

- The compositional compilation algorithm[8] 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.
- Here, 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.[1,10] An improved version of the algorithm is called CPA (Cartesian Product Algorithm).[11] In these approaches types are set of classes, like in Strongtalk,[12] a typechecker for Smalltalk. More recently, a modified CPA[5] 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 5 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 espressive; furthermore, while our system is based on type inference, in Java 5 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*[?,?] 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.

As already mentioned, type inference has been deeply investigated in the context of functional languages since the early 80s, and many of the systems proposed in literature are based on the Hindley/Milner system with constraints.[13] In particular, HM(X)[14] is a general framework for Hindley/Milner style systems with constraints, analogous to the CLP(X) framework in constraint logic programming, which also include a notion of subsumption relation and can therefore adapted to a wide variety of type systems, by instantiating the parameter X with a suitable constraint system.

6. Conclusion

We have shown that 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. 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. Even though we do not attempt at giving a precise formulation of this statement, we think that the type system proposed here is in a sense "the most flexible" one can superimpose on, say, Featherweight Java, taken as the representative of Java-like languages.

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 in our opinion 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.

As mentioned above, we plan to investigate in more detail some foundational aspects of the work presented in this paper, such as showing that our polymorphic types actually correspond to *principal typings*[15] for methods, and comparing our approach with type inference in Standard ML. Another important subject of future work is the study of the impact of our proposed extension on the various aspects of the full Java language. In particular, exception handling and overloading of polymorphic methods are two important features which are to be taken into account in order to obtain a practical extension of Java.

References

1. J. Palsberg and M. I. Schwartzbach, Object-oriented type inference, in *ACM Symp. on Object-Oriented Programming: Systems, Languages and Applications 1991*, 1991.
2. J. Eifrig, S. F. Smith and V. Trifonov, Type inference for recursively constrained types and its application to OOP, in *Mathematical Foundations of Programming Semantics*, , Electronic Notes in Theoretical Computer Science Vol. 1 (Elsevier Science, 1995).
3. J. Eifrig, S. F. Smith and V. Trifonov, Sound polymorphic type inference for objects, in *ACM Symp. on Object-Oriented Programming: Systems, Languages and Applications 1995*, , SIGPLAN Notices Vol. 30(10)1995.
4. J. Palsberg, *ACM Comput. Surv.* **28**, 358 (1996).
5. T. Wang and S. F. Smith, Precise constraint-based type inference for Java, in *ECOOP'01 - European Conference on Object-Oriented Programming*, *Lecture Notes in Computer Science* **2072** (Springer, 2001).
6. A. Igarashi, B. C. Pierce and P. Wadler, Featherweight Java: A minimal core calculus for Java and GJ, in *ACM Symp. on Object-Oriented Programming: Systems, Languages and Applications 1999*, November 1999.
7. G. Lagorio and E. Zucca, Introducing safe unknown types in Java-like languages, in *ACM Symp. on Applied Computing (SAC 2006), Special Track on Object-Oriented Programming Languages and Systems*, ed. L. Liebrock (ACM Press, 2006).
8. D. Ancona, F. Damiani, S. Drossopoulou and E. Zucca, Polymorphic bytecode: Compositional compilation for Java-like languages, in *ACM Symp. on Principles of Programming Languages 2005*, (ACM Press, January 2005).
9. A. Igarashi, B. C. Pierce and P. Wadler, *ACM Transactions on Programming Languages and Systems* **23**, 396 (2001).
10. J. Palsberg and M. I. Schwartzbach, *Object-Oriented Type Systems* (John Wiley & Sons, 1994).
11. O. Agesen, The cartesian product algorithm, in *ECOOP'05 - Object-Oriented Programming*, ed. W. Olthoff, Lecture Notes in Computer Science, Vol. 952 (Springer, 1995).
12. G. Bracha and D. Griswold, Strongtalk: Typechecking Smalltalk in a production environment, in *ACM Symp. on Object-Oriented Programming: Systems, Languages and Applications 1993*, 1993.
13. R. Milner, *Journ. of Computer and System Sciences* **17**, 348 (1978).
14. M. Odersky, M. Sulzmann and M. Wehr, *Theory and Practice of Object Systems* **5**, 35 (1999).
15. J. B. Wells, The essence of principal typings, in *International Colloquium on Automata, Languages and Programming 2002*, Lecture Notes in Computer Science(2380) (Springer, 2002).

Appendix A.

For lack of space we include some figures as an Appendix.

Figure A2 shows the rules for well-formed type environments; the functions *cname*, *dom*, *mname*, and *tvars*, whose obvious formal definitions are omitted, return the name of the declared class in a class signature, the set of declared

$$\mathsf{mt} ::= \Gamma \Rightarrow \mathtt{t}_1 \ldots \mathtt{t}_n \rightarrow \mathtt{t}$$

$$\text{(mtype-1)}\ \frac{}{\mathit{mtype}(\Delta, \mathtt{C}, \mathsf{m}) = \Gamma \Rightarrow \mathtt{t}_1 \ldots \mathtt{t}_n \rightarrow \mathtt{t}}\ \begin{array}{l} (\mathtt{C}, \_, \mathsf{mss}) \in \Delta \\ \Gamma \Rightarrow \mathsf{m}\ \mathtt{t}(\mathtt{t}_1 \ldots \mathtt{t}_n) \in \mathsf{mss} \end{array}$$

$$\text{(mtype-2)}\ \frac{\mathit{mtype}(\Delta, \mathtt{C}', \mathsf{m}) = \mathsf{mt}}{\mathit{mtype}(\Delta, \mathtt{C}, \mathsf{m}) = \mathsf{mt}}\ \begin{array}{l} (\mathtt{C}, \mathtt{C}', \mathsf{mss}) \in \Delta \\ \mathsf{m} \notin \mathsf{mss} \end{array}$$

Fig. A1. Method types

classes in a type environment (conventionally including `Object`), the name of the declared method in a method signature, and the set of type variables in a method/class signature.

$$\text{(wf-ms)}\ \frac{\vdash_{nf} \Gamma}{\Delta; \mathtt{C} \vdash \Gamma \Rightarrow \mathtt{t}\ \mathsf{m}(\mathtt{t}_1 \ldots \mathtt{t}_n)}\ \begin{array}{l} i \neq j, \mathtt{t}_i \equiv \alpha_i, \mathtt{t}_j \equiv \alpha_j \implies \alpha_i \neq \alpha_j \\ \mathtt{t}_i \equiv \mathtt{C}_i \implies \mathtt{C}_i \in \mathit{dom}(\Delta), \mathtt{t} \equiv \mathtt{C} \implies \mathtt{C} \in \mathit{dom}(\Delta) \end{array}$$

$$\text{(wf-cs)}\ \frac{\Delta; \mathtt{C} \vdash \mathsf{ms}_i\ \forall i \in 1..n}{\Delta \vdash (\mathtt{C}, \mathtt{C}', \mathsf{ms}_1 \ldots \mathsf{ms}_n)}\ \begin{array}{l} \mathtt{C} \neq \mathtt{Object}, \mathtt{C}' \in \mathit{dom}(\Delta) \\ i \neq j \implies \mathit{mname}(\mathsf{ms}_i) \neq \mathit{mname}(\mathsf{ms}_j), \mathit{tvars}(\mathsf{ms}_i) \cap \mathit{tvars}(\mathsf{ms}_j) = \emptyset \\ \Delta \not\vdash \mathtt{C}' \leq \mathtt{C} \\ \mathit{mtype}(\Delta, \mathtt{C}, \mathsf{m}) = \mathsf{mt}, \mathit{mtype}(\Delta, \mathtt{C}', \mathsf{m}) = \mathsf{mt}' \implies \Delta \vdash \mathsf{mt}' \leftarrow \mathsf{mt} \end{array}$$

$$\text{(wf-}\Delta\text{)}\ \frac{\Delta \vdash \mathsf{cs}_i\ \forall i \in 1..n}{\vdash \mathsf{cs}_1 \ldots \mathsf{cs}_n \diamond}\ i \neq j \implies \mathit{cname}(\mathsf{cs}_i) \neq \mathit{cname}(\mathsf{cs}_j), \mathit{tvars}(\mathsf{cs}_i) \cap \mathit{tvars}(\mathsf{cs}_j) = \emptyset$$

Fig. A2. Well-formed type environments

6.1. *Example of type inference*

In this section we will show how to infer polymorphic method types, illustrating how the type inference algorithm works on an example. Consider the following FJ program:

```
class C1 extends Object {
   C1 m1(C1 x,C2 y) { return x; }
}

class C2 extends C1 {
   m2(x) { return x; }
   m3(x) { return new C1().m1(x,this.m2(x)); }
}
```

$$(\gamma)\frac{}{\Delta;\Gamma\vdash\gamma}\ \gamma\in\Gamma \qquad (\emptyset)\frac{}{\Delta;\Gamma\vdash\emptyset} \qquad (\gamma\ \Gamma')\frac{\Delta;\Gamma\vdash\gamma \qquad \Delta;\Gamma\vdash\Gamma'}{\Delta;\Gamma\vdash\gamma\ \Gamma'}$$

$$(\leq\text{-refl-}\mathtt{C})\frac{}{\Delta;\Gamma\vdash\mathtt{C}\leq\mathtt{C}}\ (\mathtt{C},\_,\_)\in\Delta \qquad (\leq\text{-refl-}\alpha)\frac{}{\Delta;\Gamma\vdash\alpha\leq\alpha}$$

$$(\leq\text{-inh})\frac{}{\Delta;\Gamma\vdash\mathtt{C}\leq\mathtt{C}'}\ (\mathtt{C},\mathtt{C}',\_)\in\Delta \qquad (\leq\text{-trans})\frac{\Delta;\Gamma\vdash\mathtt{t}_1\leq\mathtt{t}_2 \qquad \Delta;\Gamma\vdash\mathtt{t}_2\leq\mathtt{t}_3}{\Delta;\Gamma\vdash\mathtt{t}_1\leq\mathtt{t}_3}$$

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

Fig. A3. Rules for entailment

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

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

$$(\mathsf{md}\text{-}\alpha)\frac{\mathsf{x}_1:\mathtt{t}_1,\ldots,\mathsf{x}_n:\mathtt{t}_n,\mathtt{this}{:}\mathtt{C}_0\vdash\mathsf{e}:\Gamma\Rightarrow\mathtt{t}}{\mathtt{C}_0\vdash\mathsf{m}(\mathtt{t}_1\ \mathsf{x}_1,\ldots,\mathtt{t}_n\ \mathsf{x}_n)\ \{\mathtt{return\ e;}\}:\Gamma\Rightarrow\mathtt{t}\ \mathsf{m}(\mathtt{t}_1\ldots\mathtt{t}_n)}$$

$$(\mathsf{md}\text{-}\mathtt{C})\frac{\mathsf{x}_1:\mathtt{t}_1,\ldots,\mathsf{x}_n:\mathtt{t}_n,\mathtt{this}{:}\mathtt{C}_0\vdash\mathsf{e}:\Gamma\Rightarrow\mathtt{t}}{\mathtt{C}_0\vdash\mathtt{C}\ \mathsf{m}(\mathtt{t}_1\ \mathsf{x}_1,\ldots,\mathtt{t}_n\ \mathsf{x}_n)\ \{\mathtt{return\ e;}\}:\Gamma,\mathtt{t}\leq\mathtt{C}\Rightarrow\mathtt{C}\ \mathsf{m}(\mathtt{t}_1\ldots\mathtt{t}_n)}$$

$$(\mathsf{x})\frac{}{\Pi\vdash\mathsf{x}:\emptyset\Rightarrow\mathtt{t}}\ \Pi(\mathsf{x})=\mathtt{t}$$

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

$$(\mathsf{new})\frac{}{\Pi\vdash\mathtt{new\ C()}:\mathtt{C}\leq\mathtt{C}\Rightarrow\mathtt{C}}$$

Fig. A4. Constraint inference

We first inspect each method in isolation, assuming fresh type variables for parameter with no explicit type annotations, and generate all the constraints (involving parameter types and other classes) needed for assigning a type to the method body.

This constraint inference process is formally described by the rules shown in Figure A4 in the Appendix.

These rules are fairly straightforward: the basic idea is that, instead of verifying that a certain constraint hold in the given environment, the constraint is simply added to the set of generated constraints.

In the example we get the following:

- $\mathtt{m1}$ has type $\emptyset \Rightarrow \mathtt{C}_1\ \mathtt{C}_2 \rightarrow \mathtt{C}_1$
- taking type variable α_2 as parameter type, $\mathtt{m2}$ has type $\emptyset \Rightarrow \alpha_2 \rightarrow \alpha_2$
- taking type variable α_3 as parameter type, $\mathtt{m3}$ has type $\mathtt{C}_1 \leq \mathtt{C}_1, \mu(\beta_3\ \mathtt{C}_2.\mathtt{m2}(\alpha_3)), \mu(\gamma_3\ \mathtt{C}_1.\mathtt{m1}(\alpha_3\ \beta_3)) \Rightarrow \alpha_3 \rightarrow \gamma_3$

In this way, we have constructed an environment Δ. Now, we try to simplify the method types we have obtained. If simplification succeeds, leading to a simplified environment Δ^{nf} which is well-formed, then the program is well-typed and has type Δ^{nf}, as shown in (P). Note the difference with the corresponding rule in Figure 3, where we had an a priori environment assigning types to methods, which was used to typecheck every class. Here, instead, each class is inspected in isolation, and simplification of the resulting environment serves to check consistency, in particular that mutual assumptions of methods are satisfied.

More in detail, we try to construct, for each method type $\Gamma \Rightarrow \mathtt{t}_1 \ldots \mathtt{t}_n \rightarrow \mathtt{t}$, a set of constraints in normal form Γ^{nf} such that $\Delta^{nf}; \Gamma^{nf} \vdash$ is equivalent (modulo substitution) to $\Delta; \Gamma \vdash$. Roughly, this means that all constraints in Γ which express requirements on existing classes can be checked in Δ, and, if the check is successful, can be eliminated; in the end the only remaining constraints are those which express requirements on the parameter types.

In the example, the first two method types are already in normal form. The type of $\mathtt{m3}$, instead, contains constraints which can be simplified in the current environment. Set Γ_3 the set of these constraints, that is,

$$\mathtt{C}_1 \leq \mathtt{C}_1, \mu(\beta_3\ \mathtt{C}_2.\mathtt{m2}(\alpha_3)), \mu(\gamma_3\ \mathtt{C}_1.\mathtt{m1}(\alpha_3\ \beta_3)).$$

The first constraint, $\mathtt{C}_1 \leq \mathtt{C}_1$ holds trivially, so we mark it (in the algorithm in Figure 4 marks are expressed by adding the constraint to done) and proceed to the next one: $\mu(\beta_3\ \mathtt{C}_2.\mathtt{m2}(\alpha_3))$. In Δ, class $\mathtt{C}_2$ has a method named $\mathtt{m2}$, with type $\emptyset \Rightarrow \alpha_2 \rightarrow \alpha_2$. We take the substitution $\sigma(\alpha_2) = \alpha_3$, $\sigma(\beta_3) = \alpha_3$. The method $\mathtt{m2}$ has no constraints, hence we get

$$\mathtt{C}_1 \leq \mathtt{C}_1{}^{\star}, \mu(\alpha_3\ \mathtt{C}_2.\mathtt{m2}(\alpha_3))^{\star}, \mu(\gamma_3\ \mathtt{C}_1.\mathtt{m1}(\alpha_3\ \alpha_3))$$

where the first two constraints are star-marked to denote that they have been already checked.

Take now the constraint $\mu(\gamma_3\ \mathtt{C}_1.\mathtt{m1}(\alpha_3\ \alpha_3))$. In Δ, class $\mathtt{C}_1$ has a method named $\mathtt{m1}$, with type $\emptyset \Rightarrow \mathtt{C}_1\ \mathtt{C}_2 \rightarrow \mathtt{C}_1$. We take the empty substitution and add to the current set the constraints $\alpha_3 \leq \mathtt{C}_1, \alpha_3 \leq \mathtt{C}_2$, hence we get

$$\mathtt{C}_1 \leq \mathtt{C}_1{}^{\star}, \mu(\alpha_3\ \mathtt{C}_2.\mathtt{m2}(\alpha_3))^{\star}, \mu(\mathtt{C}_1\ \mathtt{C}_1.\mathtt{m1}(\alpha_3\ \alpha_3))^{\star}, \alpha_3 \leq \mathtt{C}_1, \alpha_3 \leq \mathtt{C}_2$$

There are no longer constraints not in normal form to be examined, hence we get the following method type in normal form for $\mathtt{m3}$:

$\alpha_3 \leq \mathtt{C}_1, \alpha_3 \leq \mathtt{C}_2 \Rightarrow \alpha_3 \rightarrow \mathtt{C}_1$

which correctly expresses[¶] the requirements on argument types needed to safely apply the method. Note that the result type has become $\mathtt{C}_1$ as an effect of applying the substitution σ.

In order to see how recursive constraints are handled, consider the following example:

```
class C {
  m1(x) { return this.m2(x); }
  m2(x) { return this.m1(x); }
}
```

In this case, type inference rules lead to the following method types:

- taking type variable α as parameter type, `m1` has type $\mu(\beta\ \mathtt{C.m2}(\alpha)) \Rightarrow \alpha \rightarrow \beta$
- taking type variable γ as parameter type, `m2` has type $\mu(\delta\ \mathtt{C.m1}(\gamma)) \Rightarrow \gamma \rightarrow \delta$

Simplification steps of either method type, e.g., the first, are as follows. We start from

$\mu(\beta\ \mathtt{C.m2}(\alpha))$

Class `C` has a method named `m2` with type $\mu(\delta\ \mathtt{C.m1}(\gamma)) \Rightarrow \gamma \rightarrow \delta$, and we take the substitution $\sigma(\gamma) = \alpha, \sigma(\delta) = \beta$. Hence we get

$\mu(\beta\ \mathtt{C.m2}(\alpha))^\star, \mu(\beta\ \mathtt{C.m1}(\alpha))$

Now we consider the second constraint: class `C` has a method named `m1` with type $\mu(\beta\ \mathtt{C.m2}(\alpha)) \Rightarrow \alpha \rightarrow \beta$ and we take the identity substitution. We should add the constraint $\mu(\beta\ \mathtt{C.m1}(\alpha))$ which, however, is already in the set. Hence we terminate with

$\mu(\beta\ \mathtt{C.m2}(\alpha))^\star, \mu(\beta\ \mathtt{C.m1}(\alpha))^\star$

and the simplified method type is, as expected, $\emptyset \Rightarrow \alpha \rightarrow \beta$. Interestingly enough, we are able to type some recursive definitions which cannot be typed in, say, Standard ML, as in the following example[‖]

```
class D {}
class C {
  id(x) { return x; }
  m () { return id(new C()); }
  f () { return id(new D()); }
}
```

where, as the reader can easily verify, we obtain the following method types:

[¶]A smarter algorithm could further simplify this type by removing the redundant constraint $\alpha_3 \leq \mathtt{C}_1$.

[‖]An ML analogous would be: `let rec id x = x and m x = id 1 and f x = id true`.

- taking type variable α as parameter type, `id` has type $\emptyset \Rightarrow \alpha \rightarrow \alpha$
- `m` has type $\emptyset \Rightarrow \; \rightarrow \mathtt{C}$
- `f` has type $\emptyset \Rightarrow \; \rightarrow \mathtt{D}$

Of course this is possible since, roughly, we do not have higher-order features; nevertheless, we believe the result is nice in itself, also because the treatment of recursion among methods can be smoothly integrated with that of other constraints, as discussed below.