Type inference for polymorphic methods in Java-like languages

Davide Ancona, Giovanni Lagorio, and Elena Zucca

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

Abstract. In mainstream class-based object-oriented languages with nominal types, like C++, Java and C#, typechecking algorithms require methods to be annotated with their parameter types, which are either fixed or constrained by a (nominal) bound. On the contrary, languages like ML, CaML and Haskell use powerful type inference algorithms capable of calculating the type for a function in which parameter types are left unspecified. This inferred type is possibly polymorphic, hence functions can be applied to arguments of different, unrelated, types, which are instances of the same schema.

We show that, surprisingly enough, the latter scenario 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 capture the (less restrictive) requirements on arguments needed to safely apply the method. Moreover, the approach enjoys separate compilation a la Java.

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 from untyped code 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

⋆ This work has been partially supported by APPSEM II - Thematic network IST-2001-38957, and MIUR EOS - Extensible Object Systems.
A preliminary version of the ideas exploited in this paper is in [9] (see the Conclusion for a comparison).

2 A type system with polymorphic method types

We formalize our approach on a minimal language, whose syntax is given in Figure 1.

\[
P ::= cd_1 \ldots cd_n \\
cd ::= \text{class } C \text{ extends } C' \{ \text{mds} \} \quad (C \neq \text{Object}) \\
\text{mds} ::= md_1 \ldots md_n \\
md ::= mh \{ \text{return } e; \} \\
mh ::= [C] \text{ m}(t_1, x_1, \ldots, t_n, x_n) \\
t ::= C \mid \alpha \\
e ::= \text{new } C() \mid x \mid e_0.\text{m}(e_1, \ldots, e_n)
\]

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

This language is basically Featherweight Java [7], 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 \(\alpha\) (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.

```java
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 nominal
bound 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 \((\alpha, \beta)\) s.t. \(\alpha\) has a method \(m\)
applicable to \(\beta\), and the result type is that of \(m\). Formally, method `polyM` has the polymorphic
type \(\mu(\gamma \alpha. m(\beta)) \Rightarrow \alpha \beta \rightarrow \gamma\), which means that `polyM` has two parameters of type \(\alpha\) and \(\beta\) and
returns a value of type \(\gamma\) (right-hand side of \(\Rightarrow\)), providing that the constraint \(\mu(\gamma \alpha. m(\beta))\) is
satisfied (left-hand side of \(\Rightarrow\)), that is, class \(\alpha\) has a method \(m\) which can be safely applied to
an argument of type \(\beta\) by returning a value of type \(\gamma\).

In a type environment where we have this type for `Example.polyM`, typechecking of meth-
ods `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).

We will see later other examples illustrating how chains of method calls and recursion are
handled. Type environments \(\Delta\) 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 class and a
sequence of method signatures.

A method signature is a tuple consisting of a set of constraints \(\Gamma\), a result type, a method
name, and sequence of parameter types.

1 They can be easily handled by just considering new kinds of constraints, see the following.
2 We will see in Section 3 how to infer this type.
In the simple language we consider, there are only two forms of constraints: $t \leq t'$, meaning that type $t$ must be a subtype of $t'$, and $\mu(t, t_0, m(t_1 \ldots t_n))$, meaning that type $t_0$ must have a (either directly declared or inherited) method named $m$ applicable to arguments of types $t_1 \ldots t_n$ and giving, for these argument types, a result of type $t$. Fields, constructors and casts can be easily handled, as done in [2], adding constraints of the form: $\phi(t, t, f)$, meaning that type $t$ must have a (either directly declared or inherited) field named $f$ of type $t'$, $\kappa(t, t_1 \ldots t_n)$, meaning that type $t$ must have a constructor applicable to arguments of types $t_1 \ldots t_n$, and $t \sim t'$, meaning that either type $t$ must be a subtype of $t'$ 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 $\Delta$ if $\Delta$ is well-formed ($\vdash \Delta \diamond$), and every class declaration conforms to the type environment $\Delta$. The judgment $\vdash \Delta \diamond$ is defined in Figure 5 in the following.

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

Rules $(md-\alpha)$ and $(md-C)$ check that the body $e$ of $m$ conforms to the method type $\Gamma \Rightarrow t_1 \ldots t_n \rightarrow t$ found in $\Delta$, and extracted by the function $mtype$, see Figure 4. More precisely, $e$ is typechecked in $\Delta$, under the method constraints $\Gamma$, and in a parameter environment $\Pi$ which assigns to the 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 $\Delta$ $(md-C)$.

The entailment judgment $\Delta; \Gamma \vdash \gamma$ is formally defined in Figure 6 in the following. Intuitively, it holds if the constraint $\gamma$ either holds in $\Delta$ or is one of the constraints in $\Gamma$. We will also write $\Delta; \emptyset \vdash \gamma$.

The last three rules define the typing judgment for expressions, which has form $\Delta; \Pi; \Gamma \vdash e : t$. The type environment $\Delta$ is needed to perform type checks involving existing classes (for instance, $C_1 \leq C_2$), whereas the constraints $\Gamma$ express requirements on the parameter types which are just assumed to hold for the current method.
\[ \Delta \vdash \text{cd}_i : \text{cs}_i \forall i \in 1..n \quad \vdash \Delta \phi \quad \Delta = \text{cs}_1 \ldots \text{cs}_n \]

\[ \Delta; C \vdash \text{md}_i : \text{ms}_i \forall i \in 1..n \]

\[ \Delta; x_1 : t_1, \ldots, x_n : t_n, \text{this} : C_0; \Gamma \vdash e : t' \]

\[ \Delta; \Gamma \vdash \text{t}' \leq t \]

\[ \Delta; C_0 \vdash \text{m}(t_1, x_1, \ldots, t_n, x_n) \{ \text{return } e; \} : \]

\[ \Gamma \Rightarrow t \text{ m}(t_1 \ldots t_n) \]

\[ \Delta; C_0 \vdash C \text{m}(t_1, x_1, \ldots, t_n, x_n) \{ \text{return } e; \} : \]

\[ \Gamma \Rightarrow C \text{ m}(t_1 \ldots t_n) \]

Fig. 3. Rules for typechecking

\[ \text{mt ::= } \Gamma \Rightarrow t_1 \ldots t_n \rightarrow t \]

\[ \text{mtype}(\Delta, C, m) = \Gamma \Rightarrow t_1 \ldots t_n \rightarrow t \quad \Gamma \Rightarrow m \ t(t_1 \ldots t_n) \in \text{mss} \]

\[ \text{mtype}(\Delta, C, m) = \text{mt} (C, C', \text{mss}) \in \Delta \]

\[ \text{mtype}(\Delta, C, m) = \text{mt} \ m \not\in \text{mss} \]

Fig. 4. Method types
Rule (\(\pi\)) is standard.

By rule (call), in the type environment \(\Delta\) and parameter environment \(\Pi\), under the constraints \(\Gamma\), 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 arguments. 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 \(C \leq C\), 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 6).

The rules for well-formed type environments can be found in Figure 5. 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 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.

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 (\(\mu\)) in Figure 6). Constraints in method types must be in normal form, that is, of the form \(\gamma ::= \alpha \leq t \mid \mu(t \alpha.m(t_1 \ldots t_n))\); 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 7.

Figure 6 contains the formal definition of the entailment judgment. The rules are pretty straightforward, except for rule (\(\mu\)), where \(\sigma\) denotes a substitution mapping type variables into types. This rule states that a constraint \(\mu(t \ C_0.m(t_1 \ldots t_n))\) holds in a given type environment \(\Delta\), under assumptions \(\Gamma\), if in \(\Delta\) 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.
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); }
}
```

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

```
\[ \Delta \vdash \text{polyM(new A(), new A())} \]
```

in method `test` has type `C`. The invocation

```
this.m(this)
```

and, by substituting `\alpha` by `\beta` and `\gamma` with `\mu` we get the constraint `\mu(\mu(\alpha.m(\beta)))` which holds in `\Delta`.

Note that in the premise of the rule we add `\mu(t C_0.m(t_1...t_n))` to `\Gamma`. 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.m(\alpha)) \Rightarrow \alpha \rightarrow \beta`. The invocation `this.m(this)` in method `test` typechecks since the judgment `\Delta \vdash \mu(C.m(C))` holds, with `\Delta` the type environment corresponding to the program. This judgment holds since `\mu(type(\Delta, \text{Example}, \text{polyM}) = \mu(\gamma \alpha.m(\beta)) \Rightarrow \alpha \beta \gamma)` and, by substituting `\alpha`, `\beta` and `\gamma` with `\mu` we get the constraint `\mu(\mu(\alpha.m(\beta)))` which should not be proved again.

Figure 7 contains the formal definition of the overriding judgment.

\[
\text{(overriding)} \frac{\Delta, \Gamma \vdash \sigma(\Gamma')}{\Delta, \Gamma \vdash \alpha_i \equiv \sigma(\alpha_i) \Rightarrow \sigma(t_i) = t_i}
\]

Fig. 6. Rules for entailment

Fig. 7. Rule for overriding
Rule (overriding) 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 $\rightarrow_P$ are standard and reported in Figure 10 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 in [8]; roughly, in Featherweight GJ only a kind of constraints on type variables is considered, that is, that they satisfy their (recursive) upper bound.

**Theorem 1 (Progress).** If $\Delta \vdash P \text{ and } \Delta; \emptyset \vdash e : t$, then either $e = \text{new } C()$ or $e \rightarrow_P e'$ for some $e'$.

**Theorem 2 (Subject reduction).** If $\Delta \vdash P \text{ and } \Delta; \Pi; \emptyset \vdash e : t$, $e \rightarrow_P e'$, then $\Delta; \Pi; \emptyset \vdash e : t'$, $\Delta \vdash t' \leq t$.

### 3 Inferring polymorphic method types

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:

```java
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)); }
}
```

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 from second to last in Figure 8.

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:
In this way, we have constructed an environment \( \Delta \). Now, we try to simplify the method types we have obtained. If simplification succeeds, leading to a simplified environment \( \Delta^{nf} \) which is well-formed, then the program is well-typed and has type \( \Delta^{nf} \), as shown in (P). Note the difference with the corresponding rule in Figure 3, where we had an a priori environment \( \Gamma \) which express requirements on existing classes can be checked in \( \Delta \), 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.

More in detail, we try to construct, for each method type \( \Gamma \Rightarrow t_1 \ldots t_n \rightarrow t \), a set of constraints in normal form \( \Gamma^{nf} \) such that \( \Delta^{nf}; \Gamma^{nf} \vdash \) is equivalent (modulo substitution) to \( \Delta; \Gamma^{nf} \). Roughly, this means that all constraints in \( \Gamma \) which express requirements on existing classes can be checked in \( \Delta \), 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.

The algorithm which computes \( \Gamma^{nf} \) is described in pseudocode in Figure 9, together with its pre- and postcondition. The variable \( \text{all} \) contains the current set of constraints, and the variable \( \text{done} \) keeps trace of those which have already been checked. We write \( \Delta \vdash \Gamma \sim \Gamma' \) to denote that \( \Delta; \Gamma^{nf} \vdash \) and \( \Delta; \Gamma^{nf} \vdash \Gamma' \) hold.

Note that the \textbf{switch} construct covers all possible cases. Indeed, since constraints in \( \Gamma \) are generated by the type inference algorithm in Figure 8, and substitution always apply only to parameter types, it is easy to see that we never get constraints of form \( C \leq \alpha \) or \( \mu(C \cdot C.m(t_1 \ldots t_n)) \); this condition is omitted in the formal invariant for simplicity.
\begin{verbatim}
{all==l && done==[]} while (∃γ ∈ (all \ done not in normal form) && \!fail)
       done = done ∪ {γ};
switch γ
       case C ≤ C':
         if (∆\!\! \triangleright C ≤ C') failure = true;
       case μ(α C.m(t_1...t_n)):
         mt = mtype(∆, C, m);
         if (mt undefined) failure = true;
       else
         let mt = I' ⇒ t'_1...t'_m ⇒ t' in
           if (mt=μ) failure = true;
           else
             subst = {α_i → t_i | t'_i ≡ α_i};
             subst = subst ∪ {α → subst(t')};
             all = subst(all ∪ I'' \ {t_i ≤ C_i | t'_i ≡ C_i});
             done = subst(done);
       } \!\! \triangleright (\!\! \triangleright failure==((∃I''\!\! \triangleright in normal form and σ s.t. ∆ \!\! \triangleright I'' \!\! \triangleright \sim σ(I))));
\end{verbatim}

Fig. 9. Simplification of constraints

**Theorem 3 (Correctness of the algorithm).** The algorithm in Figure 9 is correct w.r.t. the given pre- and postcondition.

**Theorem 4 (Soundness of type inference).** If ⊢ cd_1...cd_n : ∆, then ∆ ⊢ cd_1...cd_n°.

**Theorem 5 (Completeness of type inference).** If P = cd_1...cd_n, ⊢ cd_i : cs_i for all i ∈ 1...n and the simplification algorithm fails on cs_1...cs_n, then there exists no ∆ s.t. ∆ ⊢ cd_1...cd_n°.

In the example, the first two method types are already in normal form. The type of m_3, instead, contains constraints which can be simplified in the current environment. Set I_3 the set of these constraints, that is,

\[ C_1 ≤ C_1, μ(β_3 C_2.m_2(α_3)), μ(γ_3 C_1.m_1(α_3 β_3)) \]

The first constraint, C_1 ≤ C_1 holds trivially, so we mark it (in the algorithm in Figure 9 marks are expressed by adding the constraint to done) and proceed to the next one: μ(β_3 C_2.m_2(α_3)). In ∆, class C_2 has a method named m_2, with type ∅ ⇒ α_2 → α_2. We take the substitution σ(α_2) = α_3, σ(β_3) = α_3. The method m_2 has no constraints, hence we get

\[ C_1 ≤ C_1, μ(α_3 C_2.m_2(α_3))^*, μ(γ_3 C_1.m_1(α_2 α_3)) \]

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

Take now the constraint μ(γ_3 C_1.m_1(α_3 α_3)). In ∆, class C_1 has a method named m_1, with type ∅ ⇒ C_1 C_2 → C_1. We take the empty substitution and add to the current set the constraints α_3 ≤ C_1, α_3 ≤ C_2, hence we get

\[ C_1 ≤ C_1, μ(α_3 C_2.m_2(α_3))^*, μ(C_1 C_1.m_1(α_3 α_3))^*, α_3 ≤ C_1, α_3 ≤ C_2 \]
There are no longer constraints not in normal form to be examined, hence we get the following method type in normal form for \( m3 \):

\[
\alpha_3 \leq C_1, \alpha_3 \leq C_2 \Rightarrow \alpha_3 \rightarrow C_1
\]

which correctly expresses\(^3\) the requirements on argument types needed to safely apply the method. Note that the result type has become \( C_1 \) as an effect of applying the substitution \( \sigma \).

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

```java
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 \( \alpha \) as parameter type, \( m1 \) has type \( \mu(\beta \ C.\ m2(\alpha)) \Rightarrow \alpha \rightarrow \beta \)
- taking type variable \( \gamma \) as parameter type, \( m2 \) has type \( \mu(\delta \ C.\ m1(\gamma)) \Rightarrow \gamma \rightarrow \delta \)

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

\[
\mu(\beta \ C.\ m2(\alpha))
\]

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

\[
\mu(\beta \ C.\ m2(\alpha))^*, \mu(\beta \ C.\ m1(\alpha))
\]

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

\[
\mu(\beta \ C.\ m2(\alpha))^*, \mu(\beta \ C.\ m1(\alpha))^*
\]

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\(^4\)

```java
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:

- taking type variable \( \alpha \) as parameter type, \( id \) has type \( \emptyset \Rightarrow \alpha \rightarrow \alpha \)
- \( m \) has type \( \emptyset \Rightarrow \rightarrow C \)
- \( f \) has type \( \emptyset \Rightarrow \rightarrow D \)

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.

\(^3\) A smarter algorithm could further simplify this type by removing the redundant constraint \( \alpha_3 \leq C_1 \).

\(^4\) An ML analogous would be: `let rec id x = x and m x = id 1 and f x = id true`. 
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 $\phi(t \ t.f)$ (type $t$ must have a field named $f$ of type $t'$) and $t \sim t'$ (type $t$ must be a subtype of $t'$ or conversely) can be handled as the $t \leq t'$ constraints, in the sense that they must be just checked, whereas constraints of the form $\kappa(t(t_1\ldots t_n))$, meaning that type $t$ must have a constructor applicable to arguments of types $t_1\ldots t_n$, are a simpler version of the $\mu(t' \ t.m(t_1\ldots t_n))$ 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 8 and 9. This prototype, written in Java, can be tried out using any Java-enabled web browser at the following URL:

http://www.disi.unige.it/person/LagorioG/justII/

Appendix D shows a screen-shot of our prototype when checking the following example:

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

At the moment of writing, it supports only the language described in the paper, so it has been a valuable acid test but nothing more. However, we are working on the extension of the supported language to include constructors, fields and some statements in order to check our approach on more significant examples.

This extension 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 to actually compile the sources into standard Java source/bytecode. The challenging part is how to translate method invocations involving polymorphic types. The problem lies in the fact that method invocations involving subexpressions having a polymorphic type cannot be directly compiled into a single JVM (Java Virtual Machine) instruction because every kind of method invocation requires to be fully annotated with the static standard types of target and arguments. Java generics, being oblivious to the JVM, are of no help.

By using reflection, as two of the authors did in [9], invocations involving polymorphic types can be easily translated into standard Java source/bytecode. Producing efficient Java bytecode, on the other hand, is more challenging.

Reflection could be probably avoided instantiating polymorphic methods into sets of monomorphic ones, à la 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 [9]. However, there we avoided the key problem of solving recursive constraint sets by imposing a rather severe restriction on polymorphic methods.
The type inference algorithm presented here can be seen as a generalization of that given in [2], for compositional compilation of Java-like languages. 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 concerning the simplification algorithm.

– The simplification algorithm in [2] only eliminates constraints, whereas here new constraints can be added since they are “inherited” from other methods invoked in a method’s body, making termination more an issue.

– Here, since we may also have type variables as method parameter types (besides as unknown result type of methods of external classes), substitutions are not necessarily ground.

A thorough comparison between the simplification algorithm presented here and SLD resolution is an interesting subject of further work.

Type inference in object oriented languages has been studied before; in particular, [14, 15] describe an algorithm for a basic language with inheritance, assignments and late-binding. An improved algorithm, the CPA (Cartesian Product Algorithm), has been described in [1]. These approaches use types which are set of classes, as does Strongtalk [3], a typechecker for Smalltalk; however, there is no type inference in Strongtalk. 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 to analyze standard Java programs (in order to statically verify some properties as downcasts correctness) rather than propose a polymorphic extension of Java.

The type system presented in this paper uses, in a sense, a mixing of nominal and structural types: indeed, basic types in the language are nominal (class names), but constraints can be seen as a (sophisticated) form of structural types. Hence, the approach is rather different w.r.t generic methods as in Java 5, which support a form of constrained polymorphism always driven by a nominal bound.

In this sense, our type constraints are more reminiscent of where-clauses [11, 4] 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 [10]. In particular, HM(X) [12] 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.

For instance, the type system of the object calculus of Eifrig et al. [5] can be defined as a particular instantiation of HM(X). However, such a calculus is based on structural subtyping
while FJ is based on nominal subtyping; therefore it remains to be seen how the approach of HM(X) is related to ours.

Furthermore, the pleasing logical interpretation of the typing rules of HM(X) suggests that a less algorithmic presentation of our type system should be matter of further investigation. For instance, the rule \((\mu)\) for entailment in Fig. 6 seems to correspond to a coinductive interpretation of the following rule

\[
\Delta \vdash \forall \bar{\alpha}. A_1, \ldots, A_n \Rightarrow A \\
\Delta \vdash \sigma A_1, \ldots, \sigma A_n
\]

where \(\Delta\) is a set of Horn clauses, \(A, A_1, \ldots, A_n\) are atoms (that is, constraints) and \(\sigma\) is a substitution. Following this approach, it should be possible to prove that our normalization \(\Delta \leadsto \Delta'\) preserves the logical interpretation of \(\Delta\), hence

\[
\Delta |\!\!\!| A \iff \Delta' |\!\!\!| A
\]

for any ground atom (that is, constraint) \(A\).

6 Conclusion

We have shown that by omitting type annotations in parameter and result types of methods and “seeing what happens” we get polymorphic methods for free. That is, an interpretation which turns out as pretty natural is to consider these methods as polymorphic, with a type expressed by a set of constraints which intuitively correspond to the minimal requirements on argument types needed to safely apply the method. Even though in this paper 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, in the sense that it types as many as possible well-behaved programs.

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. For instance, an interesting fact is that in our type system we are able to type some recursive definitions which cannot be typed in, say, Standard ML.

Moreover, the algorithm we use for constraint simplification has many analogies with SLD resolution which also we would like to investigate in the future.

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 [17] for methods, and comparing our approach with type inference in Standard ML and SLD resolution. 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


8. A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.


A  FJ Reduction rules

Fig. 10. FJ reduction rules

B  Proof of soundness of the type system

The following lemma is needed to prove progress.

**Lemma 1.** If \( \Delta \vdash P \) and \( \text{mtype}(\Delta, C, m) = \Gamma \Rightarrow t_1 \ldots t_n \rightarrow t \), then \( \text{mbody}(P, C, m) = (x_1 \ldots x_n, e_b) \) for some \( x_1 \ldots x_n, e_b \).

**Proof.** We prove the thesis by induction on the derivation of \( \text{mtype}(\Delta, C, m) \). We have applied either rule (\text{mtype-1}) or (\text{mtype-2}). In both cases \((C, \_, \text{mss}) \in \Delta\). Hence, since we have applied typing rule (\( \text{P} \)), there exists a class declaration \( \text{cd} = \text{class} \ C \ldots \{ \text{mds} \} \) in \( P \) s.t. \( \Delta \vdash \text{cd} : (C, \_, \text{mss}) \).

- Assume we have applied rule (\text{mtype-1}), hence \( \Gamma \Rightarrow t \ m(t_1 \ldots t_n) \in \text{mss} \). Since we have applied typing rule (\( \text{cd} \)), there exists method declaration \( \text{md} \equiv t \ m(t_1 x_1, \ldots, t_n x_n) \{ \text{return} e_b \} \) in \( \text{mds} \) s.t. \( \Delta; C \vdash \text{md} : \Gamma \Rightarrow t \ m(t_1 \ldots t_n) \). The thesis follows since in this case \( \text{mbody}(P, C, m) = (x_1 \ldots x_n, e_b) \).

- Assume we have applied rule (\text{mtype-2}), hence \( \text{mtype}(\Delta, C, m) = \text{mtype}(\Delta, C', m) \) and \( m \not\in \text{mss} \), hence \( m \not\in \text{mds} \). The thesis follows by inductive hypothesis since in this case \( \text{mbody}(P, C, m) = \text{mbody}(P, C', m) \). \( \square \)
Proof of Theorem 1 (Progress)

Proof. By induction on the structure of \( e \). The only case to be checked is \( e = e_0.m(e_1 \ldots e_n) \). If \( e_0 \rightarrow p e'_0 \), then we get the thesis by reduction rule (\( \text{recv} \)); otherwise, since we have applied typing rule (\( \text{call} \)), \( \Delta; \emptyset; \emptyset \vdash e_0 : t_0 \) holds, hence by inductive hypothesis \( e = \text{new} C_0().m(e_1 \ldots e_n) \) and \( t_0 \equiv C_0 \).

We must show that \( \text{mbody}(\Pi, C_0, m) = (x_1 \ldots x_n, e^b) \). Since we have applied typing rule (\( \text{call} \)), \( \Delta \vdash \mu(t, C_0.m(t_1 \ldots t_n)) \) holds. This judgment has been deduced by entailment rule (\( \mu \)), hence \( \text{mtype}(\Delta, C_0) = \Gamma \Rightarrow t'_1 \ldots t'_n \rightarrow t' \), and we conclude by Lemma 1.

Lemma 2. If \( \Delta \vdash P, \text{mtype}(\Delta, C, m) = \Gamma \Rightarrow t_1 \ldots t_n \rightarrow t \), and \( \text{mbody}(\Pi, C, m) = (x_1 \ldots x_n, e^b) \), then, for some \( C^b, t^b \),

\[
\Delta; x_1 : t_1, \ldots, x_n : t_n, \text{this}: C^b; \Gamma \vdash e^b : t^b,
\]

\( \Delta; \Gamma \vdash C \leq C^b, t^b \leq t \).

Proof. By induction on the derivation of \( \text{mbody}(\Pi, C, m) \).

- Assume we have applied rule (\( \text{mbody-1} \)), hence
  \[
  \text{class } C \text{ extends } C^b \{ \text{mds} \in P, \text{mds} \} \in P,
  \]
  \[
  t_0 : m(t_1, x_1, \ldots, t_n, x_n)\{\text{return } e^b\} \in \text{mds}
  \]

Since we have applied typing rules (\( P \)), (\( cd \)) and (\( \text{md} \)), we have that, for some \( t^b \),

\[
\Delta; x_1 : t_1, \ldots, x_n : t_n; \text{this}: C^b; \Gamma \vdash e^b : t^b
\]

\( \Delta; \Gamma \vdash t^b \leq t \)

and we can conclude by taking \( C^b = C \).

- Assume we have applied rule (\( \text{mbody-2} \)), hence
  \[
  \text{mbody}(\Pi, C, m) = \text{mbody}(\Pi, C', m),
  \]
  \[
  \text{class } C \text{ extends } C^b \{ \text{mds} \in P, \text{mds} \} \in P,
  \]
  \[
  m \notin \text{mds}
  \]

Then \( \text{mtype}(\Delta, C, m) = \text{mtype}(\Delta, C', m) \) and we can conclude by inductive hypothesis.

Lemma 3 (Type Substitution Preserves Entailment). If \( \Delta; \Gamma, \Gamma' \vdash \gamma \), and \( \Delta; \Gamma' \vdash \sigma(\Gamma') \), with variables in \( \text{dom}(\sigma) \) not in \( \Gamma \), then \( \Delta; \Gamma' \vdash \sigma(\gamma) \).

Proof. By induction on the derivation of \( \Delta; \Gamma_1, \Gamma, \Gamma_2' \vdash \gamma \).

Corollary 1 (Monotonicity of entailment). If \( \Delta; \Gamma, \Gamma' \vdash \gamma \), and \( \Delta; \Gamma' \vdash \Gamma'' \), then \( \Delta; \Gamma \vdash \gamma \).

Proof. By Lemma 3 taking the empty substitution.

Lemma 4 (Type Substitution Preserves Typing). If \( \Delta; \Pi; \Gamma, \Gamma' \vdash e : t \), and \( \Delta; \Gamma \vdash \sigma(\Gamma') \), with variables in \( \text{dom}(\sigma) \) not in \( \Gamma \), then \( \Delta; \sigma(\Pi); \Gamma \vdash e : t' \) with \( \Delta; \Gamma' \vdash t' \leq \sigma(t) \).

Proof. By induction on the derivation of \( \Delta; \Pi; \Gamma_1, \Gamma, \Gamma_2' \vdash e : t \).

Lemma 5 (Term Substitution Preserves Typing). If \( \Delta; \Pi; x_1 : t_1, \ldots, x_n : t_n; \Gamma \vdash e : t \), and for \( i \in 1 \ldots n \), \( \Delta; \Pi; \Gamma \vdash e_i : t'_i \) with \( \Delta; \Gamma' \vdash t'_i \leq t_i \), then \( \Delta; \Pi; \Gamma \vdash e[t_{1 \ldots n} / x_{1 \ldots n}] : t' \) with \( \Delta; \Gamma' \vdash t' \leq t \).

Proof. By induction on the derivation of \( \Delta; \Pi; x_1 : t_1, \ldots, x_n : t_n; \Gamma \vdash e : t \).

The following lemma holds since a well-formed type environment satisfies conditions on overriding in Figure 7.

Lemma 6. If \( \Delta \vdash \mu, \Delta \vdash \mu(t, t_0, m(t_1 \ldots t_n)) \) and \( \Delta \vdash t'_0 \leq t_0 \) hold, then \( \Delta \vdash \mu(t', t'_0, m(t_1 \ldots t_n)) \) holds for some \( t' \) s.t. \( \Delta; \Gamma' \vdash t' \leq t \).
Proof of Theorem 2 (Subject reduction)

Proof. By induction on the derivation of the judgment \( e \rightarrow_p e' \), with a case analysis on the reduction rule used.

Case call We have

\[
\text{new } C_0(\cdot, m(e_1, \ldots, e_n)) \rightarrow_p e^b[e_1 \ldots e_n/x_1 \ldots x_n][\text{new } C_0(\cdot)/\text{this}],
\]

\[
\text{body}(P, C_0, m) = (x_1 \ldots x_n, e^b)
\]

Since we have applied typing rules (call) and (new), we have

\[
\Delta; \Pi; \emptyset \vdash \text{new } C_0() : C_0,
\]
\[
\Delta; \Pi; \emptyset \vdash e_i : t_i \text{ for } i \in 1..n,
\]
\[
\Delta \vdash \mu(t C_0.m(t_1 \ldots t_n)).
\]

This last judgment has been deduced by entailment rule (\( \mu \)), hence there exist \( t^m, t'_1 \ldots t'_n, I^m, \sigma \) s.t.

\[
\text{mtype}(\Delta, C_0, m) = I^m \Rightarrow t'_1 \ldots t'_n \rightarrow t^m,
\]
\[
t'_i \equiv \alpha_i \implies \sigma(\alpha_i) = t_i,
\]
\[
\sigma(t^m) = t.
\]

By Lemma 2 we get that, for some \( C^b, t^b \)

\[
\Delta; x_1:t'_1, \ldots, x_n:t'_n, \text{this}:C^b, I^m \vdash e^b : t^b
\]
\[
\Delta; I^m \vdash C_0 \leq C^b, t^b \leq t^m.
\]

Since \( \Delta \vdash \mu(t C_0.m(t_1 \ldots t_n)) \) and \( \Delta; \mu(t C_0.m(t_1 \ldots t_n)) \vdash (\sigma(I^m)) \), by Corollary 1 we get that \( \Delta \vdash (\sigma(I^m)) \).

By Lemma 3 and 4, since \( \Delta \vdash (\sigma(I^m)) \), we get

\[
\Delta \vdash C_0 \leq C^b, \sigma(t^b) \leq \sigma(t^m) = t,
\]
\[
\Delta; x_1: \sigma(t'_1), \ldots, x_n: \sigma(t'_n), \text{this}:C^b; \emptyset \vdash e^b : \sigma(t^b).
\]

By Lemma 5

\[
\Delta; \Pi; \emptyset \vdash e^b[e_1 \ldots e_n/x_1 \ldots x_n][\text{new } C_0(\cdot)/\text{this}] : t' \text{ for } \Delta \vdash t' \leq \sigma(t^b)
\]

and we can conclude.

Case (recv) We have

\[
e_0, m(e_1, \ldots, e_n) \rightarrow_p e'_0, m(e_1, \ldots, e_n),
\]
\[
e_0 \rightarrow_p e'_0.
\]

Since we have applied typing rule (call), we have

\[
\Delta; \Pi; \emptyset \vdash e_i : t_i \text{ for } i \in 0..n,
\]
\[
\Delta \vdash \mu(t t_0.m(t_1 \ldots t_n)).
\]

By inductive hypothesis

\[
\Delta; \Pi; \emptyset \vdash e'_0 : t'_0 \text{ with } \Delta \vdash t'_0 \leq t_0
\]

By Lemma 6, we have that \( \Delta \vdash \mu(t' t'_0.m(t_1 \ldots t_n)) \) with \( \Delta \vdash t' \leq t \), hence we can conclude by applying typing rule (call).

Case (arg)

Easy. \( \square \)
C Proof of soundness and completeness of the algorithm

Proof of Theorem 3 (Correctness of the algorithm)

Proof. Take as invariant \( \gamma \)

\[
\text{!failure} \implies (\exists \sigma \mbox{ s.t. } \Delta \vdash (\text{all} \setminus \text{done}) \sim \sigma(\Gamma))
\]

\[
\& \& \text{ failure} \implies \exists \Gamma^{\infty} \text{ in normal form and } \sigma \mbox{ s.t. } \Delta \vdash \Gamma^{\infty} \sim \sigma(\Gamma)
\]

The precondition clearly implies the invariant for \( \sigma \) the identity; the conjunction of the invariant and the exit condition of the loop implies the postcondition: indeed, if \( \text{!failure} \) holds, then the postcondition follows since \( \exists \Gamma^{\infty} \) is in normal form.

We show now that \( \text{Inv} \) and the guard of the loop hold before executing the body, then \( \text{Inv} \) still holds at the end.

Assume that \( \text{Inv} \) and the guard of the loop hold, hence \( \exists \sigma \mbox{ s.t. } \Delta \vdash (\text{all} \setminus \text{done}) \sim \sigma(\Gamma) \). A constraint \( \gamma \) not in normal form is added to \( \text{done} \). There are the following cases:

\[- \gamma \equiv \text{C} \leq \text{C}', \text{ and } \Delta \not\vdash \text{C} \leq \text{C}'.
\]

In this case, \( \text{failure} \) becomes true, and it is easy to see that \( \exists \sigma \mbox{ s.t. } \Delta \vdash (\text{all} \setminus \text{done}) \sim \sigma(\Gamma) \) holds. Indeed, \( \gamma \in \sigma(\Gamma) \), and, since \( \Delta \not\vdash \text{C} \leq \text{C}' \), \( \Delta ; (\text{all} \setminus \text{done}) \vdash \gamma \) could only be proved by rule \( \gamma \) in Figure 6; but this rule cannot be applied since \( \gamma \not\in (\text{all} \setminus \text{done}) \).

\[- \gamma \equiv \text{C} \leq \text{C}', \text{ and } \Delta \vdash \text{C} \leq \text{C}'.
\]

In this case, \( \Delta ; (\text{all} \setminus \text{done}) \vdash \sigma(\Gamma) \) still holds by Corollary 1, and the converse trivially holds since we removed a constraint.

\[- \gamma \equiv \mu(\alpha \cdot \text{C.m(t}_1 \ldots \text{t}_n)) \), \text{ and mtype(\Delta, \text{C.m)} is undefined, or mtype(\Delta, \text{C.m) = C'}
\]

In this case, \( \text{failure} \) becomes true, and it is easy to see that \( \exists \sigma \mbox{ s.t. } \Delta \vdash (\text{all} \setminus \text{done}) \sim \sigma(\Gamma) \) holds. Indeed, \( \sigma(\gamma) \in \sigma(\Gamma) \), and rule \( (\mu) \) in Figure 6 is not applicable, hence \( \Delta ; (\text{all} \setminus \text{done}) \vdash \sigma(\Gamma) \) could only be proved by rule \( \gamma \) in Figure 6; but this rule cannot be applied since \( \gamma \not\in (\text{all} \setminus \text{done}) \).

\[- \gamma \equiv \mu(\alpha \cdot \text{C.m(t}_1 \ldots \text{t}_n)) \), \text{ and mtype(\Delta, \text{C.m) = C'}
\]

In this case, we construct the substitution \( \text{subst}(\alpha_i) = \text{t}_i \) for all \( \text{t}_i' \equiv \alpha_i \), and \( \text{subst}(\alpha) = \text{subst}(\text{t}') \) Moreover, \( \text{all} \setminus \text{done} \) becomes \( \text{subst}((\text{all} \setminus \text{done}) \cup \{ \sigma(\gamma) \}) \), with \( \text{all} \setminus \text{done} \) and \( \text{done} \) denoting the previous values of \( \text{all} \) and \( \text{done} \), respectively.

We have to show that \( \exists \sigma \mbox{ s.t. } \Delta \vdash \sigma(\Gamma) \). Take \( \sigma = \text{subst} \circ \sigma^{\text{old}} \), where \( \sigma^{\text{old}} \) denotes the substitution s.t. \( \Delta \vdash (\text{all} \setminus \text{done}^{\text{old}}) \sim \sigma^{\text{old}}(\Gamma) \), existing by hypothesis. Then, it is easy to see that \( \Delta \vdash (\text{all} \setminus \text{done}^{\text{old}}) \sim \sigma(\Gamma) \).

Moreover, \( \Delta ; (\text{all} \setminus \text{done}) \vdash \sigma(\gamma) \equiv \mu(\text{subst}(\text{t}') \cdot \text{C.m(t}_1 \ldots \text{t}_n)) \) can be derived by rule \( (\mu) \) for entailment, since

\[
\text{mtype(\Delta, \text{C.m) = C'}} \implies \text{t}_1' \ldots \text{t}_n' \rightarrow \text{t}'.
\]

Hence, by Lemma 3 \( \Delta ; \text{subst}(\text{all} \setminus \text{done}^{\text{old}} \cup \{ \gamma \}) \vdash \sigma(\Gamma) \) still holds.

Finally, termination of the algorithm is guaranteed by the fact that we never generate new variables, hence the number of distinct constraints which can be added at each step is finite. \( \square \)
The following lemma expresses the fact that constraints inferred in Figure 8 are those necessary and sufficient to typecheck a method body.

**Lemma 7.** $\Delta; \Pi; \Gamma \vdash e : t$ iff $\Pi \vdash e : \Gamma_e \Rightarrow t$ and $\Delta; \Gamma \vdash \Gamma_e$.

**Proof of Theorem 4 (Soundness of type inference)**

*Proof.* By Lemma 7 and Theorem 3 (case of successful termination). \qed

**Proof of Theorem 5 (Completeness of type inference)**

*Proof.* By Lemma 7 and Theorem 3 (case of failure). \qed
D  Our prototype in action

The following is a screen-shot of our prototype, with debugging enabled.

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

Normalized Env:

C -> {
    μ(d b n(b)) -> d n(b)
    μ(θ C m(C)) ; f ≤ Object -> Object test()
}

### Normalizing method C.m ###

* begin while *

**toBeProcessed** = { μ(d b n(b)) }, processed = {}

* all constraints in toBeProcessed are in normal form *

Env=C -> {
    μ(d b n(b)) -> d n(b)
    μ(θ C m(C)) ; f ≤ Object -> Object test()
}

### Normalizing method C.test ###

* begin while *

**toBeProcessed** = { f ≤ Object, μ(θ C m(C)) }, processed = {}

gamma=μ(θ C m(C))
mt=μ(d b n(b)) ; d n(b)
sigma=μ(b>C, T->d)

** gammaPrime = μ(d b n(b)) **

applying sigma...

** toBeProcessed** = { f ≤ Object, processed = μ(d b n(b)) }

** gammaPrime = μ(d b n(b)) **

adding new constraints in ( gammaPrime ) to toBeProcessed constraint μ(d C m(C)) shipped (already in processed)

* begin while *

** toBeProcessed** = { f ≤ Object, processed = μ(d C m(C)) }

* all constraints in toBeProcessed are in normal form, constraint 'd ≤ Object has been dropped because it is unreachable *
```