

Introducing safe unknown types in Java-like languages*

G. Lagorio
DISI - Università di Genova
Via Dodecaneso, 35
16146 Genova, Italy
lagorio@disi.unige.it

E. Zucca
DISI - Università di Genova
Via Dodecaneso, 35
16146 Genova, Italy
zucca@disi.unige.it

ABSTRACT

Most mainstream object-oriented languages, like C++, Java and C#, are statically typed. In recent years, untyped languages, in particular scripting languages for the web, have gained a lot of popularity notwithstanding the fact that the advantages of static typing, such as earlier detection of errors, are widely accepted. We think that one of the main reasons for their widespread adoption is that, in many situations, the ability of ignoring types can be handy to write simpler and more readable code.

We propose an extension of Java-like languages which allows developers to forget about typing in strategic places of their programs without losing type-safety. That is, we allow programmers to write simpler code without sacrificing the advantages of static typing. This is achieved by means of inferred type constraints. These constraints describe the implicit requirements on untyped code to be correctly invoked.

This flexibility comes at a cost: field accesses and method invocations on objects of unknown types are less efficient than regular field accesses and method invocations. Also, our type system is currently more restrictive than it should be; its extension is the subject of ongoing work. However, the novel approach presented here is quite interesting on its own, as it supports separate compilation and there is zero runtime overhead on code which does not take advantage of the new features.

Categories and Subject Descriptors

D.3.3 [Language Constructs and Features]: Classes and objects; D.3.4 [Processors]: Compilers

Keywords

Java-like languages, Static typing, Type-systems

*This work has been partially supported by APPSEM II - Thematic network IST-2001-38957, and MIUR EOS - Extensible Object Systems.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SAC'06 April, 23-27, 2006, Dijon, France

Copyright 2006 ACM 1-59593-108-2/06/0004 ...\$5.00.

1. INTRODUCTION

Consider the two following (sketched) class declarations:

```
class MyButton extends java.awt.Component {
    public boolean cmp(MyButton b) { ... }
    ...
}

class ASTNode {
    public boolean cmp(ASTNode n) { ... }
    ...
}
```

In this example, `MyButton` and `ASTNode` are two unrelated classes sharing a common feature: their instances can be compared invoking a method called `cmp`. Intuitively, we should be able to compare two `MyButton` instances using the same code that compares two `ASTNodes`.

The following snippet of code seems to confirm this intuition:

```
MyButton b1 = ... , b2 = ... ;
ASTNode n1 = ... , n2 = ... ;
System.out.println(b1.cmp(b2)) ;
System.out.println(n1.cmp(n2)) ;
```

This code can be successfully compiled, so we might decide to improve it factoring out the common code into a method `printCmp`. We could naively try to write something like:

```
void printCmp(x,y) {
    System.out.println(x.cmp(y)) ;
}
```

which is simple and nice, yet incorrect because we forgot to specify the type of the arguments `x` and `y`. The point is that there is no suitable type to be used there because `MyButton` and `ASTNode` share no common ancestor (except for `Object` which, lacking method `cmp`, would be of no help in this context).

Note that using a generic method would be not enough to do the trick either as we would need a type to describe “something which provides method `cmp`” anyway to type-check the method invocation inside the generic method.¹

Java generics can be used to solve this problem, but we need to write both a generic interface and a generic method, as discussed below.

¹A requirement like this is very similar to requirements that can be expressed in *PolyJ* using *where-clauses* [6, 2], although for different purposes; see the comparison in Section 3 for more details.

Three standard ways to solve this problem are to:

- add a common superclass;
- add a common interface;
- use reflection.

The first two solutions, that is, introducing a new supertype (either a superclass or a superinterface) of `MyButton` and `ASTNode` declaring a proper method `cmp`, have the advantage of preserving type safety but require the ability to modify the sources of `MyButton` and `ASTNode`.

Using reflection we can write a flexible solution which does not require having or changing the sources of `MyButton` and `ASTNode`, and which seamlessly works not only for `MyButton` and `ASTNode` but also for any type declaring a proper method `cmp`. However, there are two drawbacks in a reflection based solution: a slightly increased execution time, which does not bother us and would probably go unnoticed in most applications, and losing type-safety, which does bother us.

Before presenting our solution, which allows to obtain the flexibility of the reflection based solution without losing type-safety, let us discuss more in-depth the other viable solutions.

1.1 A type-safe solution

Introducing a common superclass for classes `MyButton` and `ASTNode` would be a design mistake in our example: class `MyButton` is a GUI element, so class `MyButton` has to extend (directly or indirectly) the standard class `Component`. Class `ASTNode` represents a node of an AST and is not supposed to extend any particular class.

Introducing a common superclass for `MyButton` and `ASTNode` in a single-inheritance language would force us to make `ASTNode` a descendant of `Component` too. Of course, there is no design reason to introduce such a relation. In this kind of scenario, the only viable option is to introduce an interface (and then to make both `MyButton` and `ASTNode` implement it).

The two methods `cmp` defined in our example classes cannot be properly described by a single interface, because their argument types differ. Anyway, a generic interface can be used to describe them both:²

```
interface ICanBeCompared<T> {
    boolean cmp(T other) ;
}
```

So, we can solve our problem making `MyButton` implement `ICanBeCompared<MyButton>`, `ASTNode` implement `ICanBeCompared<ASTNode>` and then declaring a generic method `printCmp`:

```
<T> void printCmp(ICanBeCompared<T> o1, T o2) {
    System.out.println(o1.cmp(o2)) ;
}
```

While this solution works and is the most effective regarding execution speed, it is not trivial to write and requires changing the sources of both `MyButton` and `ASTNode` which, in some situations, could be unavailable.

²This interface is indeed very similar to the standard generic interface `Comparable<T>`.

1.2 Reflection-based solution

Exploiting reflection we can write a solution which does not require any change to classes `MyButton` and `ASTNode` and that can be compiled in isolation³:

```
void printCmp(Object o1, Object o2)
    throws NoSuchMethodException,
           IllegalAccessException,
           InvocationTargetException {
    Object result = null ;
    for(Method m : o1.getClass().getMethods())
        if ( m.getName().equals("cmp") ) {
            Class<?> [] p = m.getParameterTypes() ;
            if (p.length==1 && p[0].isInstance(o2)) {
                result = m.invoke(o1, o2) ;
                break ;
            }
        }
    System.out.println(result) ;
}
```

On the one hand, the flexibility of this solution, which works for any type providing the proper method `cmp`, is unbeatable.

On the other hand, this method is not simple to write, not to mention the fact that a caller of such a method should take care of some exceptions which were not present before. Indeed, this solution corresponds in practice to replace static type checking of parameters by dynamic type checking which throws exceptions in case arguments fail to be adequate (since the parameter types of this method are as most generic as they can be).

1.3 Our proposed solution

We propose to add a feature allowing programmers to forget about typing in strategic places of their code. This untyped code results to be much simpler to write and maintain than the typed one. Sticking to our example, we propose to allow developers to specify as parameter and/or result type of a method the special *unknown type*, indicated with “?”, wherever they do not want to commit to a certain type⁴. Method `printCmp`, discussed above, can be written as:

```
void printCmp(? x, ? y) {
    System.out.println(x.cmp(y)) ;
}
```

We think that, from a programmer’s point of view, this is clearly the most natural (and easiest) solution.

Because we do not want to trade ease of coding with type-safety, we simplify the task of developers at the cost of a more complex type-checking algorithm.

Our idea is to translate methods having parameters/result of unknown types using reflection, generating, at the same time, type constraints that describe when such methods can be correctly invoked. These constraints contain type variables, in place of actual type names, to represent the unknown types.

In this particular example, method `printCmp` would be translated as described in Section 1.2 along with a type constraint with the following informal meaning: “the type of parameter `x` must provide a method called `cmp` which can be called with an argument of the type of parameter `y`”.

³Of course, inside a proper class omitted here.

⁴An alternative syntax could consist in just omitting these types.

In these settings, the correctness of an actual invocation of method `printCmp` can be checked evaluating its type constraints, after having substituted the type variables contained in the constraints with the actual types that are known in the context of the caller.

For instances, an invocation like `printCmp(new MyButton(), new MyButton())` can be proved to be type-correct, while an invocation `printCmp("hello", "world")` is rejected as type-incorrect (because type `String` does not provide a proper `cmp` method).

2. FORMALIZATION

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

This language is basically Featherweight Java [5], a tiny Java subset which has become a standard example to illustrate extensions and new technologies for Java-like languages; here we even omit fields, since they are not relevant for our aim. The only new feature we introduce is the fact that parameter and/or result types of methods can be, besides class names, type variables α .⁵

The formal definition of the typechecking and the translation from a program in this language to a program in plain Featherweight Java (that is, with no type variables) is given in Figure 2 and Figure 3, where we have used the following notation:

$$\tilde{t} = \begin{cases} c & \text{if } t = c \\ \text{Object} & \text{otherwise} \end{cases}$$

However, for lack of space, in this paper we omit the definition of some judgments and do not comment on these figures in detail, but rather illustrate the technique on an example similar to the one discussed in the introduction (adjusted to fit in Featherweight Java).

Suppose we want to compile the following program:

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

First of all, we distinguish between *polymorphic methods*, that is, methods with at least an untyped parameter, and

⁵However, as shown in the previous section, in a concrete syntax the user should either use a special symbol such as “?” or simply omit these types, and fresh type variables should be automatically generated by the compiler.

```
P ::= cd1 ... cdn
cd ::= class c extends c' { mds } (c ≠ Object)
mds ::= md1 ... mdn
md ::= mh {return e;}
mh ::= t0 m(t1 x1, ..., tn xn)
e ::= new c() | x | (c)e | e0.m(e1, ..., en)
t ::= c | α
```

where method and parameter names in `mds` and `mh` are distinct and t_0 can be α only if at least one t_i is α'

Figure 1: Syntax

standard methods (whose parameters are all of known types). In this example, method `Example.printM` 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 common signature as in Java generic methods.

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

In this paper, we consider a type system which imposes a rather severe restriction on polymorphic methods: they cannot invoke polymorphic methods on a target of a known type. This restriction ensures that we can first typecheck polymorphic methods, generating the type constraints describing the requirements on their parameters, then typecheck all standard methods. We are currently working on less restrictive type systems; see the conclusions for further comments. Also, for sake of simplicity we assume here no overriding for polymorphic methods; allowing this feature would require, roughly, to associate to a polymorphic method *all* the constraints generated for its redefined versions. Moreover, in order to achieve separate compilation, a polymorphic method should not generate constraints not implied by constraints of overridden methods, making in practice desirable programmer-defined constraints (again, see the conclusions).

The translation of a program `P`, using polymorphic methods, to a program `P'` in plain Featherweight Java occurs in four steps:

1. Extraction, from `P`, of a type environment Δ which maps class names to their types. We write $\Delta(c, m) = t_1 \dots t_n \rightarrow t$ to indicate that in Δ class `c` has a method named `m`, whose parameters have types $t_1 \dots t_n$ and whose return type is t . Any type t can be either a class name `c` or a type variable α .
2. Typechecking of all polymorphic methods with respect to type environment Δ and their declaring class `c` (needed to type `this`). The judgment

$$\Delta; c \vdash md : \Gamma \rightsquigarrow md'$$

has the following informal meaning: in type environment Δ , inside class `c`, method declaration `md` is translated to `md'` and requires type constraints Γ on the arguments to hold in order to be correctly invoked. This

$\frac{}{\Delta; \Pi \vdash \text{new } c() : c \mid \emptyset \rightsquigarrow \text{new } c()}$	$\frac{}{\Delta; \Pi \vdash x : t \mid \emptyset \rightsquigarrow x} \quad \Pi(x) = t$
$\frac{\Delta; \Pi \vdash e : c \mid \Gamma \rightsquigarrow e'}{\Delta; \Pi \vdash (c')e : c' \mid \Gamma \rightsquigarrow (c')e'} \quad \Delta \vdash c \sim c'$	$\frac{\Delta; \Pi \vdash e : \alpha \mid \Gamma \rightsquigarrow e'}{\Delta; \Pi \vdash (c)e : c \mid \Gamma \cup \{\alpha \sim c\} \rightsquigarrow (c)e'}$
$\frac{\Delta; \Pi \vdash e_0 : \alpha \mid \Gamma_0 \rightsquigarrow e'_0 \quad i \in 1..n \quad \Delta; \Pi \vdash e_i : t_i \mid \Gamma_i \rightsquigarrow e'_i}{\Delta; \Pi \vdash e_0.m(e_1, \dots, e_n) : \alpha' \mid \{\mu(\alpha, m, t_1 \dots t_n, \alpha')\} \cup (\bigcup_{i \in 0..n} \Gamma_i) \rightsquigarrow e'_0.\text{reflectiveInvoke}("m", e'_1, \dots, e'_n)}$	$\alpha' \text{ fresh}$
$\frac{\Delta; \Pi \vdash e_0 : c_0 \mid \Gamma_0 \rightsquigarrow e'_0 \quad i \in 1..n \quad \Delta; \Pi \vdash e_i : t_i \mid \Gamma_i \rightsquigarrow e'_i}{\Delta; \Pi \vdash e_0.m(e_1, \dots, e_n) : t \mid \{\alpha_i \leq c'_i \mid \alpha_i = t_i\} \cup (\bigcup_{i \in 0..n} \Gamma_i) \rightsquigarrow e'_0.m((c'_1)e'_1, \dots, (c'_n)e'_n)}$	$\Delta(c_0, m) = c'_1 \dots c'_n \rightarrow t \quad \forall i \in 1..n \quad t_i = c_i \implies \Delta \vdash c_i \leq c'_i$
$\frac{\Delta; \{\text{this} \mapsto c, x_1 \mapsto t_1, \dots, x_n \mapsto t_n\} \vdash e : c \mid \Gamma \rightsquigarrow e'}{\Delta; c \vdash c_0 \ m(t_1 \ x_1, \dots, t_n \ x_n) \ \{\text{return } e; \} : \Gamma \rightsquigarrow c_0 \ m(t_1 \ x_1, \dots, t_n \ x_n) \ \{\text{return } e'; \}} \quad \Delta \vdash c \leq c_0$	
$\frac{\Delta; \{\text{this} \mapsto c, x_1 \mapsto t_1, \dots, x_n \mapsto t_n\} \vdash e : \alpha \mid \Gamma \rightsquigarrow e'}{\Delta; c \vdash c_0 \ m(t_1 \ x_1, \dots, t_n \ x_n) \ \{\text{return } e; \} : \Gamma \cup \{\alpha \leq c_0\} \rightsquigarrow c_0 \ m(t_1 \ x_1, \dots, t_n \ x_n) \ \{\text{return } (c_0)e'; \}}$	
$\frac{\Delta; \{\text{this} \mapsto c, x_1 \mapsto t_1, \dots, x_n \mapsto t_n\} \vdash e : t \mid \Gamma \rightsquigarrow e'}{\Delta; c \vdash \alpha \ m(t_1 \ x_1, \dots, t_n \ x_n) \ \{\text{return } e; \} : \Gamma \cup \{\alpha \equiv t\} \rightsquigarrow \text{Object } m(\tilde{t}_1 \ x_1, \dots, \tilde{t}_n \ x_n) \ \{\text{return } e'; \}}$	

Figure 2: Polymorphic method typechecking and translation rules

judgment is defined in term of the judgment

$$\Delta; \Pi \vdash e : t \mid \Gamma \rightsquigarrow e'$$

which has the following informal meaning: in type environment Δ and parameter environment Π , expression e has type t and is translated to e' ; in order to be type-correct, the type constraints Γ must hold. A parameter environment Π maps parameter names to their types and **this** to the class being typed. Figure 2 contains the rules defining both these judgments, whereas judgments for classes and programs are standard, hence omitted here (see, e.g., [1]).

We model reflection in Featherweight Java in an abstract way by assuming that the predefined class **Object** has a **reflectiveInvoke** primitive method which takes a string representing a method name as first parameter (see the first rule for method call in Figure 2). The run-time behaviour of a call to this method is to invoke the corresponding method, if any, on the receiver with the given arguments, or to give a run-time error (formally, a stuck term) in case the method is absent.

3. Creating an enriched type environment ∇ , extending Δ with the type constraints generated during the previous step. That is, $\nabla(c, m)$ is equal to $\Delta(c, m)$ for standard methods, and equal to

$$\Delta(c, m), \Gamma$$

if Γ is the set of type constraints that has been generated when typing the polymorphic method $c.m$.

4. Typechecking all standard methods using the judgment

$$\nabla; c \vdash md \rightsquigarrow md'$$

with the following informal meaning: in an enriched type environment ∇ , inside class c , method declaration md is correct and translated to md' . As in the polymorphic case, this judgment is defined in term of another judgment used to type expressions, that is, $\nabla; \Pi \vdash e : t \rightsquigarrow e'$. Figure 3 contains the rules defining both these judgments.

The type environment Δ extracted from our example is the following:

$$\Delta = \left\{ \begin{array}{l} A \mapsto \{ m \mapsto (A \rightarrow A) \} \\ B \mapsto \{ m \mapsto (B \rightarrow B) \} \\ \text{Example} \mapsto \{ \text{print} \mapsto (\text{Object} \rightarrow \text{Object}) \\ \text{printM} \mapsto (\alpha_1 \ \alpha_2 \rightarrow \alpha) \\ \text{okA} \mapsto (\rightarrow \text{Object}) \\ \text{okB} \mapsto (\rightarrow \text{Object}) \\ \text{notOk} \mapsto (\rightarrow \text{Object}) \} \end{array} \right\}$$

Typechecking of **Example.printM** succeeds, generating the following type constraints:

$$\Gamma = \{ \mu(\alpha_1, m, \alpha_2, \alpha_3), \alpha_3 \leq \text{Object}, \alpha \equiv \text{Object} \}$$

The first constraint, $\mu(\alpha_1, m, \alpha_2, \alpha_3)$, is generated when typechecking the invocation $x.m(y)$ and has the following informal meaning: α_1 must provide a method named m which

$\overline{\nabla; \Pi \vdash \text{new } c() : c \rightsquigarrow \text{new } c()}$	$\overline{\nabla; \Pi \vdash x : t \rightsquigarrow x}$	$\Pi(x) = t$
$\frac{\nabla; \Pi \vdash e : c \rightsquigarrow e'}{\nabla; \Pi \vdash (c')e : c' \rightsquigarrow (c')e'}$	$\nabla \vdash c \sim c'$	
$\frac{i \in 0..n \quad \nabla; \Pi \vdash e_i : c_i \rightsquigarrow e'_i}{\nabla; \Pi \vdash e_0.m(e_1, \dots, e_n) : c \rightsquigarrow e'_0.m(e'_1, \dots, e'_n)}$	$\nabla(c_0, m) = t'_1 \dots t'_n \rightarrow c, \Gamma$ $\forall i \in 1..n \quad t'_i = c_i \implies \nabla \vdash c_i \leq c'_i$ \exists substitution $\sigma : \sigma(\alpha_i) = c_i$ for all $t'_i = \alpha_i$ $\nabla \vdash \sigma(\Gamma) \diamond$	
$\frac{i \in 0..n \quad \nabla; \Pi \vdash e_i : c_i \rightsquigarrow e'_i}{\nabla; \Pi \vdash e_0.m(e_1, \dots, e_n) : \sigma(\alpha) \rightsquigarrow (\sigma(\alpha))(e'_0.m(e'_1, \dots, e'_n))}$	$\nabla(c_0, m) = t'_1 \dots t'_n \rightarrow \alpha, \Gamma$ $\forall i \in 1..n \quad t'_i = c_i \implies \nabla \vdash c_i \leq c'_i$ \exists substitution $\sigma : \sigma(\alpha_i) = c_i$ for all $t'_i = \alpha_i$ $\nabla \vdash \sigma(\Gamma) \diamond$	
$\frac{\nabla; \{\text{this} \mapsto c, x_1 \mapsto c_1, \dots, x_n \mapsto c_n\} \vdash e : c' \rightsquigarrow e'}{\nabla; c \vdash c_0 \ m(c_1 \ x_1, \dots, c_n \ x_n) \ \{\text{return } e;\} \rightsquigarrow c_0 \ m(c_1 \ x_1, \dots, c_n \ x_n) \ \{\text{return } e';\}}$	$\nabla \vdash c' \leq c_0$	

Figure 3: Standard method typechecking and translation rules

can receive an α_2 returning an object of type α_3 . Type variable α_3 is a fresh variable generated during the typechecking and corresponds to the type of the whole method invocation.

The second constraint, $\alpha_3 \leq \text{Object}$, is generated by the invocation `this.print(x.m(y))` because the invoked method is `Example.print` and the type of its argument (α_3) must be a subtype of its parameter type (`Object`). The type of this expression is `Object` because method `print` returns an `Object`.

Finally, the variable α , which represents the return type of method `printM`, is equated⁶ to the type found typechecking its body (`Object`).

Now we can extend Δ with Γ obtaining ∇ , which is equal to Δ except for `Example.printM` where it also returns the above type constraints Γ .

Having found ∇ , we can typecheck all remaining methods. Methods `A.m`, `B.m` and `Example.print` can be trivially checked; let us directly discuss the interesting ones, starting from `Example.okA`.

In `Example.okA` we find an invocation to a polymorphic method: `printM`. In this invocation the arguments have both type `A`, so to decide whether the invocation is safe we need to check whether Γ , the constraints of `printM`, are satisfied under a suitable substitution which maps parameter type variables to their respective (known) argument types. The formal definition of the judgment $\Delta \vdash \gamma$, where γ is a constraint without type variables, is standard (see, e.g., [1]), hence omitted here; for instance, a constraint $\mu(c_0, m, c_1 \dots c_n, c)$ is satisfied whenever in Δ class c_0 has an (either directly declared or inherited) method m whose parameter types are supertypes of c_1, \dots, c_n and whose return type is c . We can find the needed substitution σ (or find out that it does not exist) by an algorithm which is essentially that described in [1]. The algorithm iterates through the constraints Γ , starting from the initial substitution of parameter type variables with their argument types. At each step, the algorithm considers a type constraint γ and, because the way constraints are processed, there are only two possible out-

⁶We use the symbol \equiv in type constraints to avoid possible confusion with the equality symbol used at the meta-level.

comes: either σ is enriched (with one or more mappings found by evaluating γ) or γ cannot be satisfied by any substitution. In the case of our example, our algorithm starts with: $\sigma = \{\alpha_1 \mapsto \mathbf{A}, \alpha_2 \mapsto \mathbf{A}\}$.

The constraints to be checked, after having applied the initial substitution, are:

$$\Gamma' = \{\mu(\mathbf{A}, m, \mathbf{A}, \alpha_3), \alpha_3 \leq \text{Object}, \alpha \equiv \text{Object}\}$$

Checking the first constraint, $\mu(\mathbf{A}, m, \mathbf{A}, \alpha_3)$ we find that it is satisfied for $\alpha_3 = \mathbf{A}$. The remaining constraints are now trivially satisfied, so the whole method invocation is safe and has type `Object` (that is, the value associated to α).

Method `Example.okB` is found to be safe with the same reasoning. Typechecking of method `Example.notOk`, on the other hand, should fail. In this case, the substitution of the argument types in Γ produces:

$$\Gamma'' = \{\mu(\mathbf{A}, m, \mathbf{B}, \alpha_3), \alpha_3 \leq \text{Object}, \alpha \equiv \text{Object}\}$$

The first constraint cannot be satisfied this time, as class `A` does not provide any method which can receive an object of type `B`, so the invocation of `printM` is correctly forbidden.

2.1 Results (sketched)

The type system guarantees that well-typed programs using polymorphic methods are translated into well-typed programs in plain Featherweight Java which, moreover, never get stuck due to a call to the `reflectiveInvoke` primitive method. The theorems given for FJ in the original paper [5] can be extended to take our extension into account. For lack of space, we just sketch the main results here.

THEOREM 1 (WELL-TYPED TRANSLATION). *If:*

- $\Delta; \Pi \vdash e : t \mid \Gamma \rightsquigarrow e'$, then $\tilde{\Delta}; \tilde{\Pi} \vdash_{\text{FJ}} e' : \tilde{t}$
- $\nabla; \Pi \vdash e : t \rightsquigarrow e'$, then $\tilde{\nabla}; \tilde{\Pi} \vdash_{\text{FJ}} e' : \tilde{t}$

where $\Delta; \Pi \vdash_{\text{FJ}} e : t$ denotes the FJ type judgment and $\tilde{\Delta}, \tilde{\Pi}, \tilde{\nabla}$ are the translation of, respectively, Δ, Π , and ∇ .

For stating the progress theorem, we recall from [5] the meaning of function `mbody`. Given a program, which remains

implicit,

$$\text{mbody}(\text{m}, \text{c}) = \text{x}_1 \dots \text{x}_n; \text{e}$$

if class c provides a method named m whose parameter names are $\text{x}_1, \dots, \text{x}_n$ and whose body is e .

THEOREM 2 (EXTENSION OF PROGRESS). *If expression e is the translation of a well-typed expression, and e includes the subexpression*

$$\text{new } \text{c}().\text{reflectiveInvoke}(\text{"m"}, \text{e}_1, \dots, \text{e}_n),$$

then $\text{mbody}(\text{m}, \text{c}) = \text{x}_1 \dots \text{x}_n; \text{e}_0$ for some $\text{x}_1, \dots, \text{x}_n$ and e_0 .

3. CONCLUSIONS

We have proposed an extension of Java-like languages enabling programmers to forget about typing in strategic places of their programs without losing type safety. The initial motivation has been allowing simpler and more maintainable code. However, using type constraints, rather than standard Java types also makes the language more flexible. Indeed, there are cases where no suitable (standard Java) types could be used in place of the unknown types, so the mechanism can do more than just making programs more concise; in essence, it supports quantification over types by means of the inferred constraints.

We achieved our goal mixing known technologies like reflection and inferred type constraints for Java-like languages [1] in a novel way. An important property of the approach presented here is the fact that there is zero runtime overhead on code which does not take advantage of the new features.

An alternative implementation technique⁷ could be an heterogeneous translation of our extension. That is, a translation where each polymorphic method is translated to a set of standard methods where the unknown types have been replaced by the types used by the various callers. This translation is indeed appealing from the standpoint of runtime efficiency, as there would be no overhead in using the new features, at the cost of a possible bloat in code size.

Another interesting alternative to be considered is allowing constraints to be explicitly written by programmers, presumably in the shape of where clauses (see below) about the parameters of polymorphic methods. Indeed, from a software engineering point of view, it is always debatable whether type inference is a good choice as there is a trade-off between conciseness and maintainability, since changes to the details of the implementation of a method might (accidentally) invalidate call sites for the method. Also, in an extension allowing overriding of polymorphic methods, this choice would allow programmers to describe constraints which are intended to hold in all future redefinitions, exactly as it happens for `throws` clauses in Java.

As already mentioned, the type system we have presented imposes a rather severe restriction on polymorphic methods, which avoids having to solve recursive constraint sets. Indeed, here we are more interested in proposing a rather lightweight extension to Java-like languages, allowing more flexibility to the programmer at the price of a relatively small complication in the type system and implementation, rather than proposing a new full polymorphic language. However, we believe this direction is very interesting as well, and we are currently working on it. Previous work on inferring type

constraints for object oriented languages includes [8, 4, 3, 7, 9].

Our type constraints are somewhat reminiscent of *where-clauses* [6, 2] 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 mentioned above, we are currently working on exploiting the ideas in this paper to get a full polymorphic language. 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 overriding of polymorphic methods are two important features which are to be taken into account in order to obtain a practical extension of Java.

Acknowledgements We warmly thank an anonymous referee for making us aware of the connection of our work with where-clauses and for his/her other useful suggestions. We also wish to thank Walter Cazzola for his feedback.

4. REFERENCES

- [1] 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*, 2005.
- [2] M. Day, R. Gruber, B. Liskov, and A. C. Meyers. Subtypes vs. where clauses: Constraining parametric polymorphism. In *OOPSLA '95 Conference Proceedings*, volume 30(10) of *ACM SIGPLAN Notices*, pages 156–168, 1995.
- [3] J. Eifrig, S. Smith, and V. Trifonov. Sound polymorphic type inference for objects. In *OOPSLA '95 Conference Proceedings*, volume 30(10) of *ACM SIGPLAN Notices*, pages 169–184, 1995.
- [4] J. Eifrig, S. Smith, and V. Trifonov. Type inference for recursively constrained types and its application to OOP. In *Mathematical Foundations of Programming Semantics, New Orleans*, volume 1 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1995. <http://www.elsevier.nl/locate/entcs/volume1.html>.
- [5] A. Igarashi, B. 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*, pages 132–146, November 1999.
- [6] A. C. Myers, J. A. Bank, and B. Liskov. Parameterized Types for Java. In *ACM Symp. on Principles of Programming Languages 1997*, pages 132–145, 1997.
- [7] J. Palsberg. Type inference for objects. *CSURV: Computing Surveys*, 28, 1996.
- [8] J. Palsberg and M. I. Schwartzbach. Object-oriented type inference. *ACM SIGPLAN Notices*, 26(11):146–161, November 1991.
- [9] T. Wang and S. F. Smith. Precise constraint-based type inference for Java. *Lecture Notes in Computer Science*, 2072:99–??, 2001.

⁷Suggested by an anonymous referee.