Journal of Object Technology

Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2002

Vol. 6, No. 2, Special Issue OOPS Track at SAC 2006, February 2007

Just: safe unknown types in Java-like languages

Giovanni Lagorio, DISI, University of Genova, Italy
Elena Zucca, DISI, University of Genova, Italy

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.
We have implemented our approach on a small, yet significant, Java subset.

1 INTRODUCTION

Consider the two following (sketched) class declarations:

```
class List {
   public void append(List list) { ... }
   public Integer size() { ... }
...
}

class Archive {
   public void append(Archive archive) { ... }
   public Long size() { ... }
...
}
```

In this example, `List` and `Archive` are two unrelated classes sharing some similarities: an instance of class `List` can be appended to another using the method `append` and, likewise, an instance of `Archive` can be appended to another archive using a method named `append` (note that, while these two methods have the same name, their parameter types differ). Analogously, the size of a `List`, an `Integer`, and the size of an `Archive`, a `Long`, can be obtained invoking a method named `size`. Intuitively, we should be able to append one object to another and then print the size of the result regardless the type of the two objects as long as both are `List`s or `Archive`s.

The following snippet of code seems to confirm this intuition:

```
List l1 = ...;
List l2 = ...;
l1.append(l2);
System.out.println( l1.size() );
Archive a1 = ...;
Archive a2 = ...;
a1.append(a2);
System.out.println( a1.size() );
```

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

```
void appAndSize (x, y) {
    x.append(y);
    System.out.println(x.size());
}
```

which is simple and nice, yet incorrect because we forgot to specify the type of the parameters `x` and `y`. The point is that there is no suitable type to be used there because `List` and `Archive` share no common ancestor (except for Object, which would be of no help in this context since it lacks methods append and size). Here we are trying to address unanticipated software evolution: of course, in consistent and well-written class libraries, like the standard Java API, useful commonalities among classes should have been already factored out in ready-to-use (generic) interfaces.

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 methods `append` and `size`" in order to typecheck 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.

Three standard ways to solve this problem are to:

- adding a common superclass;

[1]A requirement like this is very similar to requirements that can be expressed in *PolyJ* using *where-clauses* [10, 3], although for different purposes; see the comparison in Section 6 for more details.

- adding a common interface;
- using reflection.

The first two solutions, that is, introducing a new supertype (either a superclass or a superinterface) of `List` and `Archive`, declaring two proper methods `append` and `size`, have the advantage of preserving type-safety but require the ability to modify the sources of `List` and `Archive`.

By using reflection we can write a flexible solution which does not require having or changing the sources of `List` and `Archive`, and which seamlessly works not only for `List` and `Archive` but also for any type declaring the proper methods `append` and `size`. However, there are two drawbacks in a reflection-based solution: a slightly increased execution time, which would probably go unnoticed in most applications, and losing type-safety, which is a more serious problem.

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.

A type-safe solution

Introducing a common superclass for classes `List` and `Archive` would be a design mistake in our example since we assumed to deal with two unrelated classes. So, the only viable option is to introduce an interface (and then to make both `List` and `Archive` implement it).

The two methods `append` defined in our example classes cannot be properly described by a single interface, because their argument types differ (an analogous reasoning applies to methods `size`: in their case the return type differs). Anyway, a generic interface can be used to describe them both:

```
interface AppendAndSize<T, S> {
    void append(T t);
    S size();
}
```

So, we can solve our problem making `List` implement `AppendAndSize<List,Integer>`, `Archive` implement `AppendAndSize<Archive, Long>` and then declaring a generic method `appAndSize`:

```
<T, S> void appAndSize(AppendAndSize<T,S> x, T y) {
    x.append(y);
    System.out.println(x.size());
}
```

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 `List` and `Archive` which, in some situations, could be unavailable. So, this solution does not really support unanticipated evolution of software.

Reflection-based solution

By exploiting reflection we can write a solution which does not require any change to classes `List` and `Archive` and that can be compiled in isolation[2]:

```
void appAndSize(Object x, Object y) {
   invoke(x, "append", y);
   System.out.println(invoke(x, "size"));
}

static Object invoke(Object target, String name, Object ... args) {
   Class<?> [] argTypes = new Class<?> [args.length];
   for(int a=0; a<argTypes.length; ++a)
      argTypes[a]=args[a].getClass();
   outer: for(Method m : target.getClass().getMethods())
      if ( m.getName().equals(name) ) {
         Class<?> [] params = m.getParameterTypes();
         if (params.length==argTypes.length) {
            for(int i=0; i<params.length; ++i)
               if (! params[i].isInstance(argTypes[i]) )
                  continue outer;
            try  {
               return m.invoke(target, args);
            } catch (IllegalAccessException iae) {
               return null;
            } catch (InvocationTargetException ite) {
               return null;
            }
         }
      }
    return null;
}
```

In this approach the method `invoke` can be written once and for all, and the method `appAndSize` is quite readable. On the one hand, the flexibility of this solution, which works for any type providing the proper methods `append` and `size`, is unbeatable.

On the other hand, the method `invoke` is not obvious and its execution can take much more time than a standard method invocation. Furthermore, we have deliberately ignored the error conditions here, returning `null` when something goes wrong, but a real implementation should deal with them. Indeed, this solution

[2]Of course, inside a proper class omitted here.

corresponds in practice to replace static type checking of parameters by dynamic type checking (since the parameter types of this method are as most generic as they can be).

Our proposed solution

We propose to add a feature allowing programmers to forget about typing in strategic places of their code. This untyped code is 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 type[3]. Method `appAndSize`, discussed above, can be written in our language, Just (for “Java unknown safe types”), as:

```
void appAndSize (? x, ? y) {
    x.append(y);
    System.out.println(x.size());
}
```

We think that, from programmers’ 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 `void appAndSize (? x, ? y)` would be translated to method `void appAndSize (Object x, Object y)` shown when describing the reflection-based solution, along with a couple of type constraints with the following informal meaning: “the type of parameter `x` must provide a method called `append` that can be called with an argument of the type of parameter `y`, and a parameterless method `size`”.

In these settings, the correctness of an actual invocation of method `appAndSize` 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.

[3]An alternative supported syntax consists in omitting these types altogether. Note that even though the symbol “?” is used by Java wildcards, there can be no ambiguity: our “?” can be used only to mark the absence of programmer specified types, as opposed to compiler inferred types, for method parameters while Java wildcards are only used when types are indeed specified. So, technically our “?” is not a type, rather it is a way not to specify a type.

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

where method and parameter names in $\overline{\mathsf{md}}$ and mh are distinct and $\mathtt{t}_0$ can be α only if at least one $\mathtt{t}_i$ is α'

Figure 1: Syntax of Just

For instance, an invocation like `appAndSize(new List(), new List())` can be proved to be type-correct, while an invocation `appAndSize("hello", "world")` is rejected as type-incorrect (because type `String` provides neither method `append` nor `size`).

The rest of the paper is structured as follows. In Section 2 we introduce a minimal syntax for Just and informally describe how our type system works on an example. In Section 3 we give the formal definition of typechecking and translation of Just programs into plain Java programs, for which we prove the soundness in Section 4. In Section 5 we briefly describe the implementation and finally in Section 6 we outline related and further work.

This paper is an extended and improved version of [9].

2 JUST: AN INFORMAL INTRODUCTION

We illustrate our approach on a minimal syntax for Just, given in Figure 1.

This language is basically Featherweight Java [7] (shortly FJ), 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.[4]

Before giving the formal definition of the typechecking and the translation of a program in Just to a program in plain FJ, let us introduce a small example on which we will illustrate the technique. Suppose we want to compile the following program:

[4]However, as shown in the previous section, in the concrete syntax the user can either use a special symbol such as "?" or simply omit these types, and fresh type variables are automatically generated by the compiler.

```
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 a parameter of unknown type, and *standard* (or *monomorphic*) 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 intuition 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).

Note that classes can be still separately compiled as standard Java compilers do if inferred type constraints of polymorphic methods are inserted in `.class` files (they could be written as annotations, for instance). That is, to typecheck a class `C` we need to know only the type signatures of the classes used by `C` (provided that these signatures include type constraints too).

In this paper, we consider a type system that imposes a rather severe restriction on polymorphic methods: they cannot invoke other polymorphic methods. This restriction ensures that we can first typecheck polymorphic methods, generating the constraints describing the requirements on their argument types, 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. Alternatively, in order to achieve separate compilation, a polymorphic method which overrides another should generate weaker constraints, making in practice desirable programmer-declared constraints, similarly to what happens for checked exceptions in Java (again, see the conclusions).

The typechecking and translation of a program `P` consists of the following steps:

- checking the well-formedness of `P` (this step is as in plain FJ, except for the additional check that overriding of polymorphic methods is forbidden);
- typechecking and translation of polymorphic methods (in this phase, constraints are *generated*);
- typechecking and translation of monomorphic methods (in this phase, constraints are *checked*).

In our example, assuming that type variables in `Example.printM` are as follows:

$$\alpha\ \texttt{printM}\ (\alpha_1\ \texttt{x},\ \alpha_2\ \texttt{y})\ \{\ \ldots\ \ \}$$

typechecking of the method succeeds, generating the following constraints:

$$\overline{\gamma} = \{\mu(\alpha_1, \texttt{m}, \alpha_2, \alpha_3), \alpha_3 \leq \texttt{Object}, \alpha \equiv \texttt{Object}\}.$$

The first constraint, $\mu(\alpha_1, \texttt{m}, \alpha_2, \alpha_3)$, is generated when typechecking the invocation `x.m(y)` and has the following informal meaning: α_1 must provide a method named `m` which can receive an argument of type α_2 returning a result 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, that is, $\alpha_3 \leq \texttt{Object}$, 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, is equated[5] to the type found by typechecking its body, that is, `Object`.

Having found the constraints for polymorphic methods, we can now 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

[5]We use the symbol $\equiv$ in constraints to avoid possible confusion with the equality symbol used at the meta-level.

is safe we need to check whether $\overline{\gamma}$, the constraints of `printM`, are satisfied under a suitable substitution which maps parameter type variables to their respective (known) argument types. A constraint $\mu(\mathtt{c}_0, \mathtt{m}, \mathtt{c}_1 \ldots \mathtt{c}_n, \mathtt{c})$ is satisfied whenever in `P` class $\mathtt{c}_0$ has an (either directly declared or inherited) method `m` whose parameter types are supertypes of $\mathtt{c}_1, \ldots, \mathtt{c}_n$ 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 $\overline{\gamma}$, 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 of the way constraints are processed, there are only two possible outcomes: 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: $\sigma = \{\alpha_1 \mapsto \mathtt{A}, \alpha_2 \mapsto \mathtt{A}\}$.

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

$$\overline{\gamma}' = \{\mu(\mathtt{A}, \mathtt{m}, \mathtt{A}, \alpha_3), \alpha_3 \leq \mathtt{Object}, \alpha \equiv \mathtt{Object}\}$$

Checking the first constraint, $\mu(\mathtt{A}, \mathtt{m}, \mathtt{A}, \alpha_3)$, we find that it is satisfied for $\alpha_3 = \mathtt{A}$. 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 $\overline{\gamma}$ produces:

$$\overline{\gamma}'' = \{\mu(\mathtt{A}, \mathtt{m}, \mathtt{B}, \alpha_3), \alpha_3 \leq \mathtt{Object}, \alpha \equiv \mathtt{Object}\}$$

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

In Section 3 we present a formal model that makes these ideas precise and has been the basis for the implemented prototype (described in Section 5). Section 4 proves the correctness of the approach, showing that no well-typed program can get stuck due to the downcasts or `reflInvk` invocations inserted by the translation.

3 FORMALIZATION

The formal definition of the typechecking and the translation from a program in Just to a program in plain FJ (that is, with no type variables) is given from Figure 2 to Figure 6.

The typechecking and translation of a program `P` is shown in Figure 2. For simplifying the notations, a program `P` is represented here as a pair $\langle <, \mathtt{MT} \rangle$ where $<$ is the (direct) inheritance relation and `MT` is the *method table*, which maps pairs of class and method names into the corresponding method declaration (this representation

$$(\text{prog})\frac{\begin{array}{c}\mathtt{P};\mathtt{c}\vdash\mathtt{MT}(\mathtt{c},\mathtt{m}):\Gamma(\mathtt{c},\mathtt{m})\rightsquigarrow\mathtt{MT}'(\mathtt{c},\mathtt{m})\ \forall\langle\mathtt{c},\ \mathtt{m}\rangle\in\mathtt{poly}(\mathtt{MT})\\ \mathtt{P};\Gamma;\mathtt{c}\vdash\mathtt{MT}(\mathtt{c},\mathtt{m})\rightsquigarrow\mathtt{MT}'(\mathtt{c},\mathtt{m})\ \forall\langle\mathtt{c},\ \mathtt{m}\rangle\in\mathtt{mono}(\mathtt{MT})\end{array}}{\vdash\mathtt{P}\rightsquigarrow\mathtt{P}':\Gamma}\quad\begin{array}{l}\mathtt{P}=\langle<,\ \mathtt{MT}\rangle\\ \mathtt{P}'=\langle<,\ \mathtt{MT}'\rangle\\ \mathtt{wellFormed}(\mathtt{P})\\ \mathtt{def}(\mathtt{MT}')=\mathtt{def}(\Gamma)\\ \qquad\qquad=\mathtt{def}(\mathtt{MT})\\ \langle\mathtt{c},\ \mathtt{m}\rangle\in\mathtt{mono}(\mathtt{MT})\implies\\ \qquad\Gamma(\mathtt{c},\mathtt{m})=\emptyset\end{array}$$

$$\begin{array}{l}\mathtt{poly}(\mathtt{MT})=\{\langle\mathtt{c},\ \mathtt{m}\rangle\in\mathtt{def}(\mathtt{MT})\mid\mathtt{isPoly}(\mathtt{MT}(\mathtt{c},\mathtt{m}))\}\\ \mathtt{mono}(\mathtt{MT})=\mathtt{def}(\mathtt{MT})\setminus\mathtt{poly}(\mathtt{MT})\\ \mathtt{isPoly}(\mathtt{t}_0\ \mathtt{m}(\mathtt{t}_1\ \mathtt{x}_1,\ldots,\mathtt{t}_n\ \mathtt{x}_n)\ \{\ldots\})\ =\ (\exists i:\mathtt{t}_i=\alpha)\end{array}$$

Figure 2: Typechecking and translation of a program

makes sense since we have assumed that a class cannot declare two methods with the same name). We denote by `def` the domain of a mapping.

The judgment

$$\vdash\mathtt{P}\rightsquigarrow\mathtt{P}':\Gamma$$

has the following informal meaning: program `P` is translated to $\mathtt{P}'$ generating the *constraint environment* Γ for polymorphic methods. A constraint environment maps pairs of class and method names to their constraints.

First, the well-formedness of the program `P` is checked: the judgment `wellFormed(P)` consists of the following checks, whose obvious formal definition is omitted:

- the inheritance hierarchy is acyclic and has root `Object`,
- `Object` has no methods, apart from the method `reflInvk` which will be used to model reflection,
- every class name appearing anywhere in `MT` must appear in $<$,
- a polymorphic method cannot be overridden,
- a monomorphic method can be overridden by another which has the same parameter types and return type.

Polymorphic methods are typechecked first: this ensures that all constraints, needed to typecheck the method invocations of polymorphic methods, are inferred before the typechecking of monomorphic methods[6].

Constraints and entailment are defined in Figure 3. As formally expressed by the (straightforward) rules defining entailment, the first three forms of constraint

[6]Note that this splitting in two phases is only possible under the assumption that polymorphic methods cannot invoke other polymorphic methods.

$$
\begin{array}{lcl}
\overline{\gamma} & ::= & \gamma_1 \ldots \gamma_n \\
\gamma & ::= & \mathtt{t} \leq \mathtt{t}' \mid \mathtt{t} \sim \mathtt{t}' \mid \mathtt{t} \equiv \mathtt{t}' \mid \mu(\mathtt{t}, \mathtt{m}, \mathtt{t}_1 \ldots \mathtt{t}_n, \mathtt{t}')
\end{array}
$$

$$
(\leq\text{-refl})\frac{}{\mathtt{P} \vdash \mathtt{c} \leq \mathtt{c}}\ \begin{array}{l}\mathtt{P} = \langle <, \mathtt{MT} \rangle \\ \mathtt{c} < \mathtt{c}'\end{array}
\qquad
(\leq\text{-inh})\frac{}{\mathtt{P} \vdash \mathtt{c} \leq \mathtt{c}'}\ \begin{array}{l}\mathtt{P} = \langle <, \mathtt{MT} \rangle \\ \mathtt{c} < \mathtt{c}'\end{array}
$$

$$
(\leq\text{-trans})\frac{\mathtt{P} \vdash \mathtt{c}_1 \leq \mathtt{c}_2 \qquad \mathtt{P} \vdash \mathtt{c}_2 \leq \mathtt{c}_3}{\mathtt{P} \vdash \mathtt{c}_1 \leq \mathtt{c}_3}
$$

$$
(\sim\text{-up})\frac{\mathtt{P} \vdash \mathtt{c} \leq \mathtt{c}'}{\mathtt{P} \vdash \mathtt{c} \sim \mathtt{c}'}
\qquad
(\sim\text{-down})\frac{\mathtt{P} \vdash \mathtt{c}' \leq \mathtt{c}}{\mathtt{P} \vdash \mathtt{c} \sim \mathtt{c}'}
\qquad
(\equiv)\frac{}{\mathtt{P} \vdash \mathtt{c} \equiv \mathtt{c}}
$$

$$
(\mu)\frac{}{\mathtt{P} \vdash \mu(\mathtt{c}, \mathtt{m}, \mathtt{c}_1 \ldots \mathtt{c}_n, \mathtt{c}')}\ \begin{array}{l}\mathtt{monomtype}(\mathtt{P}, \mathtt{c}, \mathtt{m}) = \mathtt{c}'_1 \ldots \mathtt{c}'_n \rightarrow \mathtt{c}' \\ \forall i \in 1..n\ \mathtt{P} \vdash \mathtt{c}_i \leq \mathtt{c}'_i\end{array}
$$

Figure 3: Constraints and entailment

mean that $\mathtt{t}$ and $\mathtt{t}'$ are, respectively, in subtyping relation, comparable, and exactly the same type; the last form means that type $\mathtt{t}$ must provide method $\mathtt{m}$ applicable to arguments of types $\mathtt{t}_1, \ldots, \mathtt{t}_n$ with result type $\mathtt{t}'$.

The phase of typechecking and translation of polymorphic methods is described in Figure 4, and formally consists of two judgments:

- $\mathsf{P}; \mathsf{c} \vdash \mathsf{md} : \overline{\gamma} \leadsto \mathsf{md}'$, with informal meaning: in program P, inside class c (needed to type `this`), method declaration md is well-typed and is translated to md'; in order to be correctly invoked, constraints $\overline{\gamma}$ (on the argument types) must hold.

- $\mathsf{P}; \Pi \vdash \mathsf{e} : \mathtt{t} \,|\, \overline{\gamma} \leadsto \mathsf{e}'$, with informal meaning: in program P and parameter environment Π, expression e has type $\mathtt{t}$ and is translated to e'; in order to be type-correct, the constraints $\overline{\gamma}$ must hold. A parameter environment Π maps parameter names to their types and `this` to the class being typed.

The constraints found in this step are assembled into the constraint environment Γ, that is used in the following step to typecheck the invocations of polymorphic methods.

The judgment $\mathsf{P}; \mathsf{c} \vdash \mathsf{md} : \overline{\gamma} \leadsto \mathsf{md}'$ is defined by the last three rules in Figure 4. The first one, (kkMeth), models the case in which both the type of the body and the return type are classes, hence the former must be a subtype of the latter. The second one, (kuMeth), models the case in which the type of the body is unknown, that is, is a type variable α, whereas the return type is a class $\mathtt{c}_0$: in this case, a constraint is added expressing the fact that α must be a subtype of $\mathtt{c}_0$ in each invocation of the method. Moreover, a cast is inserted since in the translation the body will have type `Object`. Finally, the last one, (uMeth), models the case in

$$\text{(new)}\ \dfrac{}{\mathtt{P};\Pi\vdash\mathtt{new\ c()} : \mathtt{c}\mid\emptyset\rightsquigarrow\mathtt{new\ c()}}$$

$$\text{(param)}\ \dfrac{}{\mathtt{P};\Pi\vdash\mathtt{x} : \mathtt{t}\mid\emptyset\rightsquigarrow\mathtt{x}}\quad \Pi(\mathtt{x})=\mathtt{t}$$

$$\text{(kCast)}\ \dfrac{\mathtt{P};\Pi\vdash\mathtt{e} : \mathtt{c}\mid\overline{\gamma}\rightsquigarrow\mathtt{e}'}{\mathtt{P};\Pi\vdash(\mathtt{c}')\mathtt{e} : \mathtt{c}'\mid\overline{\gamma}\rightsquigarrow(\mathtt{c}')\mathtt{e}'}\quad \mathtt{P}\vdash\mathtt{c}\sim\mathtt{c}'$$

$$\text{(uCast)}\ \dfrac{\mathtt{P};\Pi\vdash\mathtt{e} : \alpha\mid\overline{\gamma}\rightsquigarrow\mathtt{e}'}{\mathtt{P};\Pi\vdash(\mathtt{c})\mathtt{e} : \mathtt{c}\mid\overline{\gamma}\cup\{\alpha\sim\mathtt{c}\}\rightsquigarrow(\mathtt{c})\mathtt{e}'}$$

$$\text{(uInvk)}\ \dfrac{\begin{array}{l}\mathtt{P};\Pi\vdash\mathtt{e}_0 : \alpha\mid\overline{\gamma_0}\rightsquigarrow\mathtt{e}'_0\\ \mathtt{P};\Pi\vdash\mathtt{e}_i : \mathtt{t}_i\mid\overline{\gamma_i}\rightsquigarrow\mathtt{e}'_i\ \forall i\in 1..n\end{array}}{\begin{array}{l}\mathtt{P};\Pi\vdash\ \mathtt{e}_0.\mathtt{m}(\mathtt{e}_1,\ldots,\mathtt{e}_n) : \alpha'\mid\{\mu(\alpha,\mathtt{m},\mathtt{t}_1\ldots\mathtt{t}_n,\alpha')\}\cup(\bigcup_{i\in 0..n}\overline{\gamma_i})\rightsquigarrow\\ \qquad\mathtt{e}'_0.\mathtt{reflInvk}(\text{"}\mathtt{m}\text{"},\mathtt{e}'_1,\ldots,\mathtt{e}'_n)\end{array}}$$
$$\alpha'\text{ fresh}$$

$$\text{(kInvk)}\ \dfrac{\begin{array}{l}\mathtt{P};\Pi\vdash\mathtt{e}_0 : \mathtt{c}_0\mid\overline{\gamma_0}\rightsquigarrow\mathtt{e}'_0\\ \mathtt{P};\Pi\vdash\mathtt{e}_i : \mathtt{t}_i\mid\overline{\gamma_i}\rightsquigarrow\mathtt{e}'_i\ \forall i\in 1..n\end{array}}{\begin{array}{l}\mathtt{P};\Pi\vdash\ \mathtt{e}_0.\mathtt{m}(\mathtt{e}_1,\ldots,\mathtt{e}_n) : \mathtt{c}\mid\{\alpha_i\leq\mathtt{c}'_i\mid\alpha_i=\mathtt{t}_i\}\cup(\bigcup_{i\in 0..n}\overline{\gamma_i})\\ \qquad\rightsquigarrow\mathtt{e}'_0.\mathtt{m}((\mathtt{c}'_1)\mathtt{e}'_1,\ldots,(\mathtt{c}'_n)\mathtt{e}'_n)\end{array}}$$
$$\begin{array}{l}\mathtt{monomtype}(\mathtt{P},\mathtt{c}_0,\mathtt{m})=\\ \ \mathtt{c}'_1\ldots\mathtt{c}'_n\rightarrow\mathtt{c}\\ \forall i\in 1..n\ \mathtt{t}_i=\mathtt{c}_i\implies\\ \qquad\mathtt{P}\vdash\mathtt{c}_i\leq\mathtt{c}'_i\end{array}$$

$$\text{(kkMeth)}\ \dfrac{\mathtt{P};\{\mathtt{this}\mapsto\mathtt{c},\mathtt{x}_1\mapsto\mathtt{t}_1,\ldots,\mathtt{x}_n\mapsto\mathtt{t}_n\}\vdash\mathtt{e} : \mathtt{c}'\mid\overline{\gamma}\rightsquigarrow\mathtt{e}'}{\begin{array}{l}\mathtt{P};\mathtt{c}\vdash\ \mathtt{c}_0\ \mathtt{m}(\mathtt{t}_1\ \mathtt{x}_1,\ldots,\mathtt{t}_n\ \mathtt{x}_n)\ \{\ \mathtt{return\ e};\ \} : \overline{\gamma}\rightsquigarrow\\ \qquad\mathtt{c}_0\ \mathtt{m}(\widetilde{\mathtt{t}_1}\ \mathtt{x}_1,\ldots,\widetilde{\mathtt{t}_n}\ \mathtt{x}_n)\ \{\ \mathtt{return\ e}';\ \}\end{array}}\quad \mathtt{P}\vdash\mathtt{c}'\leq\mathtt{c}_0$$

$$\text{(kuMeth)}\ \dfrac{\mathtt{P};\{\mathtt{this}\mapsto\mathtt{c},\mathtt{x}_1\mapsto\mathtt{t}_1,\ldots,\mathtt{x}_n\mapsto\mathtt{t}_n\}\vdash\mathtt{e} : \alpha\mid\overline{\gamma}\rightsquigarrow\mathtt{e}'}{\begin{array}{l}\mathtt{P};\mathtt{c}\vdash\ \mathtt{c}_0\ \mathtt{m}(\mathtt{t}_1\ \mathtt{x}_1,\ldots,\mathtt{t}_n\ \mathtt{x}_n)\ \{\ \mathtt{return\ e};\ \} : \overline{\gamma}\cup\{\alpha\leq\mathtt{c}_0\}\rightsquigarrow\\ \qquad\mathtt{c}_0\ \mathtt{m}(\widetilde{\mathtt{t}_1}\ \mathtt{x}_1,\ldots,\widetilde{\mathtt{t}_n}\ \mathtt{x}_n)\ \{\ \mathtt{return}\ (\mathtt{c}_0)\mathtt{e}';\ \}\end{array}}$$

$$\text{(uMeth)}\ \dfrac{\mathtt{P};\{\mathtt{this}\mapsto\mathtt{c},\mathtt{x}_1\mapsto\mathtt{t}_1,\ldots,\mathtt{x}_n\mapsto\mathtt{t}_n\}\vdash\mathtt{e} : \mathtt{t}\mid\overline{\gamma}\rightsquigarrow\mathtt{e}'}{\begin{array}{l}\mathtt{P};\mathtt{c}\vdash\ \alpha\ \mathtt{m}(\mathtt{t}_1\ \mathtt{x}_1,\ldots,\mathtt{t}_n\ \mathtt{x}_n)\ \{\ \mathtt{return\ e};\ \} : \overline{\gamma}\cup\{\alpha\equiv\mathtt{t}\}\rightsquigarrow\\ \qquad\mathtt{Object}\ \mathtt{m}(\widetilde{\mathtt{t}_1}\ \mathtt{x}_1,\ldots,\widetilde{\mathtt{t}_n}\ \mathtt{x}_n)\ \{\ \mathtt{return\ e}';\ \}\end{array}}$$

Figure 4: Typechecking and translation of polymorphic methods

which the return type is unknown, that is, is a variable α; in this case, a constraint is added expressing the fact that α will be equal to the type $\mathtt{t}$ of the body in each invocation of the method.

In all three cases, unknown parameter types (that is, type variables) are replaced by `Object` in the translated code. We have used the notation:

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

The judgment $\mathtt{P}; \Pi \vdash \mathtt{e} : \mathtt{t} \,|\, \overline{\gamma} \rightsquigarrow \mathtt{e}'$ is defined by the first six rules in Figure 4. The most interesting rules are the last two of the group, (uInvk) and (kInvk), that model method invocations.

The former models the case in which the target has an unknown type, that is, a type variable α. In this case, a constraint is added expressing the fact that (each instantiation of) α must provide an applicable method, and the invocation is translated by using reflection. We model reflection in FJ in an abstract way by assuming that the predefined class `Object` has a `reflInvk` primitive method which takes a string representing a method name as first parameter. 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. The resulting type of the method invocation will be in turn an unknown type, that is, a fresh type variable α'.

The latter rule models the case in which the type of the target is a class. In this case, this class must provide an applicable monomorphic method (indeed, recall that invocations of polymorphic methods inside polymorphic methods are forbidden). The (standard) definition of monomorphic method types `mmt` and of the function `monomtype` which returns the monomorphic method type associated to a pair of class and method names, if any, is given in Figure 5.

Note that applicable means, in this case, that whenever the argument type is a class $\mathtt{c}_i$, the corresponding parameter type $\mathtt{c}'_i$ must be a supertype; instead, when the argument type is unknown, that is, is a type variable α_i, a constraint is added expressing the fact that (each instantiation of) α_i must be a subtype of $\mathtt{c}'_i$. Moreover, a cast is inserted[7] since in the translation the argument will have type `Object`. The cast will always succeed since the method invocation will only be executed when the constraints hold.

The phase of typechecking and translation of monomorphic methods is described in Figure 6. The judgment

$$\mathsf{P}; \Gamma; \mathsf{c} \vdash \mathsf{md} \rightsquigarrow \mathsf{md}'$$

has the following informal meaning: in the program $\mathtt{P}$ and constraint environment Γ (piecewise found in the previous step for polymorphic methods), inside class $\mathtt{c}$, method declaration md is correct and translated to md'. As in the polymorphic case,

[7]In the rule casts are inserted for all arguments for simplicity.

$$
\begin{array}{rcl}
\mathsf{mmt} & ::= & \mathtt{c}_1 \ldots \mathtt{c}_n \rightarrow \mathtt{c} \\
\mathsf{mt} & ::= & \overline{\gamma} \Rightarrow \mathtt{t}_1 \ldots \mathtt{t}_n \rightarrow \mathtt{t}
\end{array}
$$

$$
\text{(monomtype-1)}\,\frac{}{\mathtt{monomtype}(\mathtt{P},\mathtt{c},\mathtt{m}) = \mathtt{c}_1 \ldots \mathtt{c}_n \rightarrow \mathtt{c}_0}
\quad
\begin{array}{l}
\mathtt{P} = \langle <, \mathtt{MT} \rangle \\
\mathtt{MT}(\mathtt{c},\mathtt{m}) = \\
\quad \mathtt{c}_0\ \mathtt{m}(\mathtt{c}_1\ \mathtt{x}_1, \ldots, \mathtt{c}_n\ \mathtt{x}_n)\ \{\ldots\}
\end{array}
$$

$$
\text{(monomtype-2)}\,\frac{\mathtt{monomtype}(\mathtt{P},\mathtt{c}',\mathtt{m}) = \mathsf{mmt}}{\mathtt{monomtype}(\mathtt{P},\mathtt{c},\mathtt{m}) = \mathsf{mmt}}
\quad
\begin{array}{l}
\mathtt{P} = \langle <, \mathtt{MT} \rangle \\
\langle \mathtt{c},\ \mathtt{m} \rangle \notin \mathtt{def}(\mathtt{MT}) \\
\mathtt{c} < \mathtt{c}'
\end{array}
$$

$$
\text{(mtype-1)}\,\frac{}{\mathtt{mtype}(\mathtt{P},\Gamma,\mathtt{c},\mathtt{m}) = \overline{\gamma} \Rightarrow \mathtt{t}_1 \ldots \mathtt{t}_n \rightarrow \mathtt{t}}
\quad
\begin{array}{l}
\mathtt{P} = \langle <, \mathtt{MT} \rangle \\
\mathtt{MT}(\mathtt{c},\mathtt{m}) = \\
\quad \mathtt{t}_0\ \mathtt{m}(\mathtt{t}_1\ \mathtt{x}_1, \ldots, \mathtt{t}_n\ \mathtt{x}_n)\ \{\ldots\} \\
\Gamma(\mathtt{c},\mathtt{m}) = \overline{\gamma}
\end{array}
$$

$$
\text{(mtype-2)}\,\frac{\mathtt{mtype}(\mathtt{P},\Gamma,\mathtt{c}',\mathtt{m}) = \mathsf{mt}}{\mathtt{mtype}(\mathtt{P},\Gamma,\mathtt{c},\mathtt{m}) = \mathsf{mt}}
\quad
\begin{array}{l}
\mathtt{P} = \langle <, \mathtt{MT} \rangle \\
\langle \mathtt{c},\ \mathtt{m} \rangle \notin \mathtt{def}(\mathtt{MT}) \\
\mathtt{c} < \mathtt{c}'
\end{array}
$$

Figure 5: Monomorphic and polymorphic method types

$$
\text{(new)}\,\frac{}{\mathtt{P};\Gamma;\Pi \vdash \mathtt{new\ c}() : \mathtt{c} \leadsto \mathtt{new\ c}()}
\qquad
\text{(param)}\,\frac{}{\mathtt{P};\Gamma;\Pi \vdash \mathtt{x} : \mathtt{c} \leadsto \mathtt{x}}
\quad \Pi(\mathtt{x}) = \mathtt{c}
$$

$$
\text{(cast)}\,\frac{\mathtt{P};\Gamma;\Pi \vdash \mathtt{e} : \mathtt{c} \leadsto \mathtt{e}'}{\mathtt{P};\Gamma;\Pi \vdash (\mathtt{c}')\mathtt{e} : \mathtt{c}' \leadsto (\mathtt{c}')\mathtt{e}'}
\quad \mathtt{P} \vdash \mathtt{c} \sim \mathtt{c}'
$$

$$
\text{(invk)}\,\frac{\mathtt{P};\Gamma;\Pi \vdash \mathtt{e}_i : \mathtt{c}_i \leadsto \mathtt{e}'_i\ \forall i \in 0..n}{\begin{array}{l}\mathtt{P};\Gamma;\Pi \vdash\ \mathtt{e}_0.\mathtt{m}(\mathtt{e}_1, \ldots, \mathtt{e}_n) : \sigma(\mathtt{t}) \leadsto \\ \qquad (\sigma(\mathtt{t}))\,(\mathtt{e}'_0.\mathtt{m}(\mathtt{e}'_1, \ldots, \mathtt{e}'_n))\end{array}}
\quad
\begin{array}{l}
\mathtt{mtype}(\mathtt{P},\Gamma,\mathtt{c}_0,\mathtt{m}) = \overline{\gamma} \Rightarrow \mathtt{t}_1 \ldots \mathtt{t}_n \rightarrow \mathtt{t} \\
\forall i \in 1..n\ \mathtt{P} \vdash \mathtt{c}_i \leq \sigma(\mathtt{t}_i) \\
\mathtt{P} \vdash \sigma(\overline{\gamma})
\end{array}
$$

$$
\text{(meth)}\,\frac{\mathtt{P};\Gamma;\{\mathtt{this} \mapsto \mathtt{c}, \mathtt{x}_1 \mapsto \mathtt{c}_1, \ldots, \mathtt{x}_n \mapsto \mathtt{c}_n\} \vdash \mathtt{e} : \mathtt{c}' \leadsto \mathtt{e}'}{\begin{array}{l}\mathtt{P};\Gamma;\mathtt{c} \vdash\ \mathtt{c}_0\ \mathtt{m}(\mathtt{c}_1\ \mathtt{x}_1, \ldots, \mathtt{c}_n\ \mathtt{x}_n)\ \{\ \mathtt{return\ e};\ \} \leadsto \\ \qquad \mathtt{c}_0\ \mathtt{m}(\mathtt{c}_1\ \mathtt{x}_1, \ldots, \mathtt{c}_n\ \mathtt{x}_n)\ \{\ \mathtt{return\ e}';\ \}\end{array}}
\quad \mathtt{P} \vdash \mathtt{c}' \leq \mathtt{c}_0
$$

Figure 6: Typechecking and translation of standard methods

this judgment is defined in term of another judgment used to type expressions, that is, $\mathtt{P};\Gamma;\Pi\vdash\mathtt{e}:\mathtt{c}\rightsquigarrow\mathtt{e}'$. Figure 6 contains the rules defining both these judgments.

These rules are standard rules for typechecking FJ expressions, except in one case, that is, in method invocation (rule (invk)). In this rule, σ denotes a substitution mapping type variables into class names. The method invocation is correct if the following conditions hold:

- There exists a method for the given target type and name (first side condition). Since this method can be polymorphic, its method type also keeps track of the constraint sequence associated to the method in Γ (empty if the method is monomorphic). The formal definition of (possibly polymorphic) method types `mt` and of the function `mtype` which returns the method type associated to a pair of class and method names, if any, is given in Figure 5.
- There exists a substitution (mapping the type variables possibly present in the method type into classes) such that the method turns out to be applicable to the invocation, that is, argument types are subtypes of parameter types (second side condition), and, moreover, constraints turn out to be satisfied (third side condition).

In this case, the resulting type of the method invocation is obtained by applying the substitution to the return type of the method `t`; moreover, a cast needs to be inserted since, if `t` is a type variable, then it is translated to `Object`. Note that, if `t` is a class already, then $\sigma(\mathtt{t})$ gives `t` and the cast is redundant (in the rule we do not distinguish this case for simplicity).

The (standard) definition of satisfaction of constraints is given in Figure 3. Note that a μ constraint is satisfied only if there is a *monomorphic* method applicable to the invocation; this is due again to the fact that we forbid invocations of polymorphic methods inside polymorphic methods, as illustrated by the following example. Consider

```
class Self{
 ? apply(? x){return x.apply(x);}
}

class Main{
 void main(){
  Self x=new Self();
  x.apply(x);
 }
}
```

Typechecking of class `Self` succeeds and method type $\mu(\alpha,\mathtt{apply},\alpha,\beta)\Rightarrow\alpha\rightarrow\beta$ is associated to method `apply` (which corresponds to the lambda-term $\lambda x.xx$). However, invocation `x.apply(x)` in method `main` is not correct, since invoking method

`Self.apply` with an argument of type `Self` would lead to a recursive call of the same method. Formally, in order to typecheck invocation `x.apply(x)`, a constraint $\mu(\texttt{Self}, \texttt{apply}, \texttt{Self}, \ldots)$ should be satisfied, and this is not the case since method `Self.apply` is polymorphic.

4 RESULTS

The soundness of our approach is expressed by two results. The former states that well-typed Just programs are translated into well-typed FJ programs, hence guarantees that Java code generated by the Just translator can be successfully compiled by a standard Java compiler. The latter states that no run-time errors are introduced by the translation. That is, execution of downcasts and `reflInvk` invocations inserted by the translation always succeeds.

In order to formally express these results, first of all let us introduce notations for typechecking-only judgments: we write $\vdash \mathtt{P} : \Gamma$ for $\vdash \mathtt{P} \leadsto \mathtt{P}' : \Gamma$, $\mathtt{P}; \Gamma; \mathtt{c} \vdash \mathtt{md}$ for $\mathtt{P}; \Gamma; \mathtt{c} \vdash \mathtt{md} \leadsto \mathtt{md}'$, and $\mathtt{P}; \Gamma; \Pi \vdash \mathtt{e} : \mathtt{c}$ for $\mathtt{P}; \Gamma; \Pi \vdash \mathtt{e} : \mathtt{c} \leadsto \mathtt{e}'$.

It is easy to see that on monomorphic programs (that is, when $\Gamma = \emptyset$) these judgments are just a rephrasing of the original FJ type system. Hence, we can express that well-typed Just programs are translated into well-typed FJ programs as follows.

Theorem 1 (Translation preserves well-typedness) *If* $\vdash P \leadsto \widetilde{P} : \Gamma$ *then* $\vdash \widetilde{P} : \emptyset$.

Proof:
Easy induction on the typing rules, proving analogous properties for method declarations and expressions. Namely, assuming $\vdash \mathtt{P} \leadsto \widetilde{\mathtt{P}} : \Gamma$, for method declarations we prove:

- if $\mathtt{P}; \mathtt{c} \vdash \mathtt{md} : \overline{\gamma} \leadsto \widetilde{\mathtt{md}}$, then $\widetilde{\mathtt{P}}; \emptyset; \mathtt{c} \vdash \widetilde{\mathtt{md}}$
- if $\mathtt{P}; \Gamma; \mathtt{c} \vdash \mathtt{md} \leadsto \widetilde{\mathtt{md}}$, then $\widetilde{\mathtt{P}}; \emptyset; \mathtt{c} \vdash \widetilde{\mathtt{md}}$.

For expressions, assuming $\widetilde{\Pi}(\mathtt{x}) = \widetilde{\Pi(\mathtt{x})}$, we prove:

- if $\mathtt{P}; \Pi \vdash \mathtt{e} : \mathtt{t} \,|\, \overline{\gamma} \leadsto \widetilde{\mathtt{e}}$, then $\widetilde{\mathtt{P}}; \emptyset; \widetilde{\Pi} \vdash \widetilde{\mathtt{e}} : \widetilde{\mathtt{t}}$
- if $\mathtt{P}; \Gamma; \Pi \vdash \mathtt{e} : \mathtt{c} \leadsto \widetilde{\mathtt{e}}$, then $\widetilde{\mathtt{P}}; \emptyset; \widetilde{\Pi} \vdash \widetilde{\mathtt{e}} : \mathtt{c}$.

These properties can be easily proved since, briefly, on monomorphic code types are preserved, and on polymorphic code type variables are mapped into `Object`, in such a way that `reflInvk` invocations and casts inserted by the translation are trivially well-typed. □

$$\text{(cast)}\,\frac{}{(\mathtt{c}')\mathtt{new\ c}() \rightarrow_{\mathtt{P}} \mathtt{new\ c}()}\ \mathtt{P} \vdash \mathtt{c} \leq \mathtt{c}'$$

$$\text{(invk)}\,\frac{}{\mathtt{new\ c}().\mathtt{m}(\mathtt{e}_1, \ldots, \mathtt{e}_n) \rightarrow_{\mathtt{P}} \mathtt{e}[\mathtt{e}_i/\mathtt{x}_i][\mathtt{new\ c}()/\mathtt{this}]}\ \begin{array}{l}\mathtt{m} \neq \mathtt{reflInvk}\\ \mathtt{mbody}(\mathtt{P}, \mathtt{c}, \mathtt{m}) =\\ \quad\langle \mathtt{x}_1, \ldots, \mathtt{x}_n,\ \mathtt{e}\rangle\end{array}$$

$$\text{(refl)}\,\frac{}{\mathtt{new\ c}().\mathtt{reflInvk}(\text{”}\mathtt{m}\text{”}, \mathtt{e}_1, \ldots, \mathtt{e}_n) \rightarrow_{\mathtt{P}} \mathtt{e}[\mathtt{e}_i/\mathtt{x}_i][\mathtt{new\ c}()/\mathtt{this}]}\ \begin{array}{l}\mathtt{mbody}(\mathtt{P}, \mathtt{c}, \mathtt{m}) =\\ \langle \mathtt{x}_1, \ldots, \mathtt{x}_n,\ \mathtt{e}\rangle\end{array}$$

$$\text{(mbody-1)}\,\frac{}{\mathtt{mbody}(\mathtt{P}, \mathtt{c}, \mathtt{m}) = \langle \mathtt{x}_1, \ldots, \mathtt{x}_n,\ \mathtt{e}\rangle}\ \begin{array}{l}\mathtt{P} = \langle <,\ \mathtt{MT}\rangle\\ \mathtt{MT}(\mathtt{c}, \mathtt{m}) =\\ \quad \mathtt{t}_0\ \mathtt{m}(\mathtt{t}_1\ \mathtt{x}_1, \ldots, \mathtt{t}_n\ \mathtt{x}_n)\ \{\mathtt{return\ e;}\}\end{array}$$

$$\text{(mbody-2)}\,\frac{\mathtt{mbody}(\mathtt{P}, \mathtt{c}', \mathtt{m}) = \langle \mathtt{x}_1, \ldots, \mathtt{x}_n,\ \mathtt{e}\rangle}{\mathtt{mbody}(\mathtt{P}, \mathtt{c}, \mathtt{m}) = \langle \mathtt{x}_1, \ldots, \mathtt{x}_n,\ \mathtt{e}\rangle}\ \begin{array}{l}\mathtt{P} = \langle <,\ \mathtt{MT}\rangle\\ \langle \mathtt{c},\ \mathtt{m}\rangle \notin \mathtt{def}(\mathtt{MT})\\ \mathtt{c} < \mathtt{c}'\end{array}$$

Figure 7: Reduction rules

In order to state the safety result, first of all we give in Figure 7 the reduction rules for Just programs. They are standard FJ reduction rules (obvious propagation rules are omitted), extended with a rule for `reflInvk` invocations which, as expected, behave as regular invocations. Note that, whereas for regular method invocations the standard FJ type system guarantees that a method is always found, invocations of `reflInvk` are potentially unsafe, and the same holds for casts.

In order to express that a program is *safe*, we introduce auxiliary judgments $\overset{\mathsf{safe}}{\vdash}$ for programs, method declarations and expressions which are inductively defined exactly as those we have previously defined, except that only upcasts are allowed and `reflInvk` invocations are typed as they were regular method invocations. In Figure 8, Figure 9 and Figure 10 we give the rules for expressions which are different from those in Figure 4 and in Figure 6 (rules for programs and method declarations do not change). The fact that only upcasts are allowed is modeled by changing the side condition in the corresponding rules. The fact that `reflInvk` invocations are typed as they were regular method invocations is modeled by adding a side condition $\mathtt{m} \neq \mathtt{reflInvk}$ in rules for method invocation and adding ad-hoc rules for `reflInvk` invocations

Now, the fact that the translation does not introduce run-time errors can be expressed as follows. Assume to typecheck and translate a safe program `P`, getting a translated program $\widetilde{\mathtt{P}}$ and constraints Γ. Take a (ground) expression `e` which can be proved to be safe by using the original type information (indeed, in the

$$(\text{safe-kcast})\ \frac{\mathtt{P};\Pi \overset{\mathsf{safe}}{\vdash} \mathsf{e} : \mathsf{c} \mid \overline{\gamma} \leadsto \mathsf{e}'}{\mathtt{P};\Pi \overset{\mathsf{safe}}{\vdash} (\mathsf{c}')\mathsf{e} : \mathsf{c}' \mid \overline{\gamma} \leadsto (\mathsf{c}')\mathsf{e}'}\ \mathtt{P}\vdash \mathsf{c} \leq \mathsf{c}'$$

$$(\text{safe-uInvk})\ \frac{\begin{array}{c}\mathtt{P};\Pi \overset{\mathsf{safe}}{\vdash} \mathsf{e}_0 : \alpha \mid \overline{\gamma_0} \leadsto \mathsf{e}'_0 \\ \mathtt{P};\Pi \overset{\mathsf{safe}}{\vdash} \mathsf{e}_i : \mathtt{t}_i \mid \overline{\gamma_i} \leadsto \mathsf{e}'_i\ \forall i \in 1..n\end{array}}{\begin{array}{ll}\mathtt{P};\Pi \overset{\mathsf{safe}}{\vdash} & \mathsf{e}_0.\mathtt{m}(\mathsf{e}_1,\ldots,\mathsf{e}_n) : \alpha' \mid \{\mu(\alpha,\mathtt{m},\mathtt{t}_1\ldots\mathtt{t}_n,\alpha')\} \cup (\bigcup_{i\in 0..n}\overline{\gamma_i}) \leadsto \\ & \mathsf{e}'_0.\mathtt{reflInvk}(\text{"}\mathtt{m}\text{"},\mathsf{e}'_1,\ldots,\mathsf{e}'_n)\end{array}}$$

$$\begin{array}{l}\alpha' \text{ fresh} \\ \mathtt{m} \neq \mathtt{reflInvk}\end{array}$$

$$(\text{safe-kInvk})\ \frac{\begin{array}{c}\mathtt{P};\Pi \overset{\mathsf{safe}}{\vdash} \mathsf{e}_0 : \mathsf{c}_0 \mid \overline{\gamma_0} \leadsto \mathsf{e}'_0 \\ \mathtt{P};\Pi \overset{\mathsf{safe}}{\vdash} \mathsf{e}_i : \mathtt{t}_i \mid \overline{\gamma_i} \leadsto \mathsf{e}'_i\ \forall i \in 1..n\end{array}}{\begin{array}{ll}\mathtt{P};\Pi \overset{\mathsf{safe}}{\vdash} & \mathsf{e}_0.\mathtt{m}(\mathsf{e}_1,\ldots,\mathsf{e}_n) : \mathsf{c} \mid \{\alpha_i \leq \mathsf{c}'_i \mid \alpha_i = \mathtt{t}_i\} \cup (\bigcup_{i\in 0..n}\overline{\gamma_i}) \\ & \leadsto \mathsf{e}'_0.\mathtt{m}((\mathsf{c}'_1)\mathsf{e}'_1,\ldots,(\mathsf{c}'_n)\mathsf{e}'_n)\end{array}}$$

$$\begin{array}{l}\mathtt{monomtype}(\mathtt{P},\mathsf{c}_0,\mathtt{m}) = \\ \ \mathsf{c}'_1\ldots\mathsf{c}'_n \to \mathsf{c} \\ \forall i \in 1..n\ \mathtt{t}_i = \mathsf{c}_i \implies \\ \qquad \mathtt{P}\vdash \mathsf{c}_i \leq \mathsf{c}'_i \\ \mathtt{m} \neq \mathtt{reflInvk}\end{array}$$

Figure 8: Safe expressions (polymorphic methods) - part 1

$$\text{(safe-uInvk)}\ \frac{\begin{array}{c}\mathtt{P};\Pi \overset{\mathsf{safe}}{\vdash} \mathtt{e}_0 : \alpha \,|\, \overline{\gamma_0} \leadsto \mathtt{e}'_0 \\ \mathtt{P};\Pi \overset{\mathsf{safe}}{\vdash} \mathtt{e}_i : \mathtt{t}_i \,|\, \overline{\gamma_i} \leadsto \mathtt{e}'_i\ \forall i \in 1..n\end{array}}{\begin{array}{ll}\mathtt{P};\Pi \overset{\mathsf{safe}}{\vdash} & \mathtt{e}_0.\mathtt{m}(\mathtt{e}_1,\ldots,\mathtt{e}_n) : \alpha' \,|\, \{\mu(\alpha,\mathtt{m},\mathtt{t}_1\ldots\mathtt{t}_n,\alpha')\} \cup (\bigcup_{i\in 0..n}\overline{\gamma_i}) \leadsto \\ & \mathtt{e}'_0.\mathtt{reflInvk}(\text{"m"},\mathtt{e}'_1,\ldots,\mathtt{e}'_n)\end{array}}$$

α' fresh

$$\text{(safe-krefl)}\ \frac{\begin{array}{c}\mathtt{P};\Pi \overset{\mathsf{safe}}{\vdash} \mathtt{e}_0 : \mathtt{c}_0 \,|\, \overline{\gamma_0} \leadsto \mathtt{e}'_0 \\ \mathtt{P};\Pi \overset{\mathsf{safe}}{\vdash} \mathtt{e}_i : \mathtt{t}_i \,|\, \overline{\gamma_i} \leadsto \mathtt{e}'_i\ \forall i \in 1..n\end{array}}{\begin{array}{ll}\mathtt{P};\Pi \overset{\mathsf{safe}}{\vdash} & \mathtt{e}_0.\mathtt{reflInvk}(\text{"m"},\mathtt{e}_1,\ldots,\mathtt{e}_n) : \mathtt{c} \,|\, \{\alpha_i \le \mathtt{c}'_i | \alpha_i = \mathtt{t}_i\} \cup (\bigcup_{i\in 0..n}\overline{\gamma_i}) \leadsto \\ & \mathtt{e}'_0.\mathtt{reflInvk}(\text{"m"},(\mathtt{c}'_1)\mathtt{e}'_1,\ldots,(\mathtt{c}'_n)\mathtt{e}'_n)\end{array}}$$

$\mathtt{monomtype}(\mathtt{P},\mathtt{c}_0,\mathtt{m}) =\ \mathtt{c}'_1\ldots\mathtt{c}'_n \rightarrow \mathtt{c}$
$\forall i \in 1..n\ \mathtt{t}_i = \mathtt{c}_i \implies \mathtt{P} \vdash \mathtt{c}_i \le \mathtt{c}'_i$

$$\text{(safe-urefl)}\ \frac{\begin{array}{c}\mathtt{P};\Pi \overset{\mathsf{safe}}{\vdash} \mathtt{e}_0 : \alpha \,|\, \overline{\gamma_0} \leadsto \mathtt{e}'_0 \\ \mathtt{P};\Pi \overset{\mathsf{safe}}{\vdash} \mathtt{e}_i : \mathtt{t}_i \,|\, \overline{\gamma_i} \leadsto \mathtt{e}'_i\ \forall i \in 1..n\end{array}}{\begin{array}{ll}\mathtt{P};\Pi \overset{\mathsf{safe}}{\vdash} & \mathtt{e}_0.\mathtt{reflInvk}(\text{"m"},\mathtt{e}_1,\ldots,\mathtt{e}_n) : \alpha' \,|\, \{\mu(\alpha,\mathtt{m},\mathtt{t}_1\ldots\mathtt{t}_n,\alpha')\} \cup (\bigcup_{i\in 0..n}\overline{\gamma_i}) \\ & \leadsto \mathtt{e}'_0.\mathtt{reflInvk}(\text{"m"},\mathtt{e}'_1,\ldots,\mathtt{e}'_n)\end{array}}$$

α' fresh

Figure 9: Safe expressions (polymorphic methods) - part 2

$$\text{(safe-cast)}\ \frac{\mathtt{P};\Gamma;\Pi \overset{\mathsf{safe}}{\vdash} \mathtt{e} : \mathtt{c} \leadsto \mathtt{e}'}{\mathtt{P};\Gamma;\Pi \overset{\mathsf{safe}}{\vdash} (\mathtt{c}')\mathtt{e} : \mathtt{c}' \leadsto (\mathtt{c}')\mathtt{e}'}\quad \mathtt{P} \vdash \mathtt{c} \le \mathtt{c}'$$

$$\text{(safe-invk)}\ \frac{\mathtt{P};\Gamma;\Pi \overset{\mathsf{safe}}{\vdash} \mathtt{e}_i : \mathtt{c}_i \leadsto \mathtt{e}'_i\ \forall i \in 0..n}{\begin{array}{ll}\mathtt{P};\Gamma;\Pi \overset{\mathsf{safe}}{\vdash} & \mathtt{e}_0.\mathtt{m}(\mathtt{e}_1,\ldots,\mathtt{e}_n) : \sigma(\mathtt{t}) \leadsto \\ & (\sigma(\mathtt{t}))\,(\mathtt{e}'_0.\mathtt{m}(\mathtt{e}'_1,\ldots,\mathtt{e}'_n))\end{array}}\quad \begin{array}{l}\mathtt{mtype}(\mathtt{P},\Gamma,\mathtt{c}_0,\mathtt{m}) = \overline{\gamma} \Rightarrow \mathtt{t}_1\ldots\mathtt{t}_n \rightarrow \mathtt{t} \\ \forall i \in 1..n\ \mathtt{P} \vdash \mathtt{c}_i \le \sigma(\mathtt{t}_i) \\ \mathtt{P} \vdash \sigma(\overline{\gamma}) \\ \mathtt{m} \neq \mathtt{reflInvk}\end{array}$$

$$\text{(safe-refl)}\ \frac{\mathtt{P};\Gamma;\Pi \overset{\mathsf{safe}}{\vdash} \mathtt{e}_i : \mathtt{c}_i \leadsto \mathtt{e}'_i\ \forall i \in 0..n}{\begin{array}{ll}\mathtt{P};\Gamma;\Pi \overset{\mathsf{safe}}{\vdash} & \mathtt{e}_0.\mathtt{reflInvk}(\text{"m"},\mathtt{e}_1,\ldots,\mathtt{e}_n) : \sigma(\mathtt{t}) \leadsto \\ & (\sigma(\mathtt{t}))\,(\mathtt{e}'_0.\mathtt{reflInvk}(\text{"m"},\mathtt{e}'_1,\ldots,\mathtt{e}'_n))\end{array}}\quad \begin{array}{l}\mathtt{mtype}(\mathtt{P},\Gamma,\mathtt{c}_0,\mathtt{m}) = \\ \qquad \overline{\gamma} \Rightarrow \mathtt{t}_1\ldots\mathtt{t}_n \rightarrow \mathtt{t} \\ \forall i \in 1..n\ \mathtt{P} \vdash \mathtt{c}_i \le \sigma(\mathtt{t}_i) \\ \mathtt{P} \vdash \sigma(\overline{\gamma})\diamond\end{array}$$

Figure 10: Safe expressions (monomorphic methods)

translation some type information is lost, since type variables are always translated to `Object`). Then, execution of `e` never gets stuck. This can be expressed by the standard progress (Theorem 3) and subject reduction (Theorem 6) properties. The proof schema is similar to that given in [8].

Before giving the formal statements and proofs of these theorems, let us briefly illustrate their meaning on the following program `P` consisting of the three classes introduced in Section 2:

```
P ≡
class A {
   A m(A anA) { return anA; }
}
class B {
   B m(B aB) { return aB; }
}
class Example {
   Object print(Object o) { return this; }
   α printM(α₁ x,α₂ y) {
       return this.print(x.m(y));
   }
}
```

The translation $\widetilde{\texttt{P}}$ and the generated constraint Γ are the following:

```
P̃ ≡
class A {
   A m(A anA) { return anA; }
}
class B {
   B m(B aB) { return aB; }
}
class Example {
   Object print(Object o) { return this; }
   Object printM(Object x,Object y) {
       return this.print(x.reflInvk("m",y));
   }
}
```

$$\begin{array}{l}\Gamma \equiv \\ \langle \texttt{A}, \texttt{m}\rangle \mapsto \emptyset \\ \langle \texttt{B}, \texttt{m}\rangle \mapsto \emptyset \\ \langle \texttt{Example}, \texttt{print}\rangle \mapsto \emptyset \\ \langle \texttt{Example}, \texttt{printM}\rangle \mapsto \{\mu(\alpha_1, \texttt{m}, \alpha_2, \alpha_3), \alpha_3 \leq \texttt{Object}, \alpha \equiv \texttt{Object}\}\end{array}$$

Program `P` is clearly well-typed (judgment $\vdash \texttt{P} : \Gamma$ holds[8]). However, program `P`

[8]Recall that this is an abbreviation for $\vdash \texttt{P} \leadsto \widetilde{\texttt{P}} : \Gamma$ for some $\widetilde{\texttt{P}}$.

is more than well-typed, is *safe* in the sense that no run-time errors can be raised by its execution (judgment $\overset{\mathsf{safe}}{\vdash} \mathtt{P} : \Gamma$ holds). This is trivially the case since `P` does not contain downcasts or `reflInvk` invocations.

Assume now to reduce a (ground) expression `e` in the context of the translated program, say $\mathtt{e} \equiv$ `new Example().printM(new A(), new A())`. It is easy to see that reduction $\rightarrow_{\widetilde{\mathtt{P}}}$ of `e` does not get stuck since, intuitively, constraints on method `printM` are satisfied. Formally, `e` can be proved to be safe w.r.t. the original type information `P` and Γ ($\mathtt{P}; \Gamma; \emptyset \overset{\mathsf{safe}}{\vdash} \mathtt{e} : \mathtt{Object}$ holds). Notably, we can only assign type `Object` to the expression `new A().reflInvk("m",new A())` by using the type information from the translated program $\widetilde{\mathtt{P}}$ and $\emptyset$, and this does not guarantee absence of run-time errors. However, by using the original type information `P` and Γ, where method `printM` has type $\{\mu(\alpha_1, \mathtt{m}, \alpha_2, \alpha_3), \alpha_3 \leq \mathtt{Object}, \alpha \equiv \mathtt{Object}\} \Rightarrow \alpha_1\alpha_2 \rightarrow \alpha$, we can prove that the same expression is safe and has type `A` since constraints hold under a suitable substitution. Theorem 3 and Theorem 6 state that this implies that execution of `e` does not get stuck.

The following lemma is needed to prove progress.

Lemma 2 *If* $\overset{\mathsf{safe}}{\vdash} P \leadsto \widetilde{P} : \Gamma$*, and* $\mathtt{mtype}(P, \Gamma, c, m) = \overline{\gamma} \Rightarrow t_1, \ldots, t_n \rightarrow t$*, then* $\mathtt{mbody}(\widetilde{P}, c, m) = \langle \mathsf{x}_1, \ldots, \mathsf{x}_n, \, \mathsf{e} \rangle$.

Proof:
We prove the thesis by induction on the derivation of $\mathtt{mtype}(\mathtt{P}, \Gamma, \mathtt{c}, \mathtt{m})$, with $\mathtt{P} = \langle <, \mathtt{MT} \rangle$, $\widetilde{\mathtt{P}} = \langle <, \widetilde{\mathtt{MT}} \rangle$.

- Assume we have applied typing rule (mtype-1), then
  $\mathtt{MT}(\mathtt{c}, \mathtt{m}) = \mathtt{t}_0\ \mathtt{m}(\mathtt{t}_1\ \mathsf{x}_1, \ldots, \mathtt{t}_n\ \mathsf{x}_n)\ \{\ldots\}$. The thesis follows since in this case $\mathtt{mbody}(\mathtt{P}, \mathtt{c}, \mathtt{m}) = \langle \mathsf{x}_1, \ldots, \mathsf{x}_n, \, \ldots \rangle$, and it is immediate to see that if $\mathtt{mbody}(\mathtt{P}, \mathtt{c}, \mathtt{m}) = \langle \mathsf{x}_1, \ldots, \mathsf{x}_n, \, \ldots \rangle$ then $\mathtt{mbody}(\widetilde{\mathtt{P}}, \mathtt{c}, \mathtt{m}) = \langle \mathsf{x}_1, \ldots, \mathsf{x}_n, \, \ldots \rangle$. $\square$

- Assume we have applied rule (mtype-2), then
  $\mathtt{mtype}(\mathtt{P}, \Gamma, \mathtt{c}, \mathtt{m}) = \mathtt{mtype}(\mathtt{P}, \Gamma, \mathtt{c}', \mathtt{m})$, with $\langle \mathtt{c}, \mathtt{m} \rangle \not\in \mathtt{def}(\mathtt{MT})$, $\mathtt{P} \vdash \mathtt{c} \leq \mathtt{c}'$. The thesis follows by inductive hypothesis since in this case it is immediate to see that $\langle \mathtt{c}, \mathtt{m} \rangle \not\in \mathtt{def}(\widetilde{\mathtt{MT}})$ and $\widetilde{\mathtt{P}} \vdash \mathtt{c} \leq \mathtt{c}'$, hence $\mathtt{mbody}(\widetilde{\mathtt{P}}, \mathtt{c}, \mathtt{m}) = \mathtt{mbody}(\widetilde{\mathtt{P}}, \mathtt{c}', \mathtt{m})$. $\square$

Theorem 3 (Progress) *If* $\overset{\mathsf{safe}}{\vdash} P \leadsto \widetilde{P} : \Gamma$*, and* $P; \Gamma; \emptyset \overset{\mathsf{safe}}{\vdash} \mathtt{e} : c$*, then either* $\mathtt{e} = \mathit{new}\ c()$ *or* $\mathtt{e} \rightarrow_{\widetilde{P}} \mathtt{e}'$ *for some* $\mathtt{e}'$.

Proof:
By induction on the structure of `e`. The cases to be checked are:

$(\mathtt{c}')\mathtt{e}$ If $\mathtt{e} \rightarrow_{\widetilde{\mathtt{P}}} \mathtt{e}'$, then we get the thesis by propagation; otherwise, since we have applied typing rule (safe-cast), $\mathtt{P}; \Gamma; \emptyset \overset{\mathsf{safe}}{\vdash} \mathtt{e} : \mathtt{c}$ holds, hence by inductive hypothesis $\mathtt{e} = \mathtt{new}\ \mathtt{c}()$. In order to apply reduction rule (cast), we must show that $\widetilde{\mathtt{P}} \vdash \mathtt{c} \leq \mathtt{c}'$, and it is immediate to see that this follows from the side condition $\mathtt{P} \vdash \mathtt{c} \leq \mathtt{c}'$ of the typing rule (safe-cast). $\square$

$\mathtt{e}_0.\mathtt{m}(\mathtt{e}_1, \ldots, \mathtt{e}_n)$, $\mathtt{m} \neq \mathtt{reflInvk}$ If $\mathtt{e}_0 \rightarrow_{\widetilde{\mathtt{P}}} \mathtt{e}_0'$, then we get the thesis by propagation; otherwise, since we have applied typing rule (safe-invk), $\mathtt{P}; \Gamma; \emptyset \overset{\mathsf{safe}}{\vdash} \mathtt{e}_0 : \mathtt{c}_0$ holds, hence by inductive hypothesis $\mathtt{e}_0 = \mathtt{new}\ \mathtt{c}_0()$. In order to apply reduction rule (invk), we must show that $\mathtt{mbody}(\widetilde{\mathtt{P}}, \mathtt{c}_0, \mathtt{m}) = \langle \mathtt{x}_1, \ldots, \mathtt{x}_n, \mathtt{e} \rangle$. Since we have applied typing rule (safe-invk), $\mathtt{mtype}(\mathtt{P}, \Gamma, \mathtt{c}_0, \mathtt{m}) = \overline{\gamma} \Rightarrow \mathtt{t}_1, \ldots, \mathtt{t}_n \rightarrow \mathtt{t}$, and we conclude by Lemma 2. $\square$

$\mathtt{e}_0.\mathtt{reflInvk}(\text{"}\mathtt{m}\text{"}, \mathtt{e}_1, \ldots, \mathtt{e}_n)$ The proof is analogous to the previous case, considering typing rule (safe-refl) and reduction rule (refl). $\square$

The following lemmas are needed to prove subject reduction.

Lemma 4 *If* $\overset{\mathsf{safe}}{\vdash} P \rightsquigarrow \widetilde{P} : \Gamma$, $\mathtt{mtype}(P, \Gamma, c, m) = \overline{\gamma} \Rightarrow t_1, \ldots, t_n \rightarrow t$, $\mathtt{mbody}(\widetilde{P}, c, m) = \langle \mathtt{x}_1, \ldots, \mathtt{x}_n, \mathtt{e} \rangle$, *and* $P \vdash \sigma(\overline{\gamma})$, *then, for some* t^b, c^b,

$P; \Gamma; \{this \mapsto c^b, \mathtt{x}_1 \mapsto \sigma(t_1), \ldots, \mathtt{x}_n \mapsto \sigma(t_n)\} \overset{\mathsf{safe}}{\vdash} \mathtt{e} : t^b$; *moreover,* $P \vdash c \leq c^b$, *and* $P \vdash t^b \leq \sigma(t)$.

Lemma 5 (Term substitution) *If* $P; \Gamma; \Pi, \mathtt{x}_1 \mapsto c_1, \ldots, \mathtt{x}_n \mapsto c_n \overset{\mathsf{safe}}{\vdash} \mathtt{e} : c$, *and for* $i \in 1..n$ $P; \Gamma; \Pi \overset{\mathsf{safe}}{\vdash} \mathtt{e}_i : c_i'$ *with* $P \vdash c_i' \leq c_i$, *then* $P; \Gamma; \Pi \overset{\mathsf{safe}}{\vdash} \mathtt{e}[\mathtt{e}_i/\mathtt{x}_i] : c'$ *with* $P \vdash c' \leq c$.

Theorem 6 (Subject reduction) *If* $\overset{\mathsf{safe}}{\vdash} P \rightsquigarrow \widetilde{P} : \Gamma$, $P; \Gamma; \Pi \overset{\mathsf{safe}}{\vdash} \mathtt{e} : c$, *and* $\mathtt{e} \rightarrow_{\widetilde{P}} \mathtt{e}'$, *then* $P; \Gamma; \Pi \overset{\mathsf{safe}}{\vdash} \mathtt{e}' : c'$ *with* $P \vdash c' \leq c$.

Proof:
By induction on the derivation of the judgment $\mathtt{e} \rightarrow_{\widetilde{\mathtt{P}}} \mathtt{e}'$, with a case analysis on the reduction rule used. We show one case.

(safe-invk) We have

$$\begin{array}{l}\mathtt{new}\ \mathtt{c}().\mathtt{m}(\mathtt{e}_1, \ldots, \mathtt{e}_n) \rightarrow_{\widetilde{\mathtt{P}}} \mathtt{e}[\mathtt{e}_i/\mathtt{x}_i][\mathtt{new}\ \mathtt{c}()/\mathtt{this}],\\ \mathtt{m} \neq \mathtt{reflInvk},\\ \mathtt{mbody}(\widetilde{\mathtt{P}}, \mathtt{c}, \mathtt{m}) = \langle \mathtt{x}_1, \ldots, \mathtt{x}_n, \mathtt{e} \rangle.\end{array}$$

Moreover, since we have applied typing rules (safe-invk) and (new), we have

$\mathtt{P};\Gamma;\Pi \overset{\textsf{safe}}{\vdash} \mathtt{new\ c().m(e_1,\ldots,e_n)} : \sigma(\mathtt{t})$
$\mathtt{P};\Gamma;\Pi \overset{\textsf{safe}}{\vdash} \mathtt{new\ c()} : \mathtt{c}$
$\mathtt{P};\Gamma;\Pi \overset{\textsf{safe}}{\vdash} \mathtt{e}_i : \mathtt{c}_i\ \forall i \in 1..n$
$\mathtt{mtype}(\mathtt{P},\Gamma,\mathtt{c},\mathtt{m}) = \overline{\gamma} \Rightarrow \mathtt{t}_1 \ldots \mathtt{t}_n \rightarrow \mathtt{t}$
$\forall i \in 1..n\ \mathtt{P} \vdash \mathtt{c}_i \leq \sigma(\mathtt{t}_i)$
$\mathtt{P}\vdash\sigma(\overline{\gamma})$

By Lemma 4, we get that, for some $\mathtt{c}^b, \mathtt{t}^b$

$\mathtt{P};\Gamma;\{\mathtt{this} \mapsto \mathtt{c}^b, \mathtt{x}_1 \mapsto \sigma(\mathtt{t}_1),\ldots,\mathtt{x}_n \mapsto \sigma(\mathtt{t}_n)\} \overset{\textsf{safe}}{\vdash} \mathtt{e} : \mathtt{t}^b$, and
$\mathtt{P}\vdash\mathtt{t}^b \leq \sigma(\mathtt{t})$.

By Lemma 5

$\mathtt{P};\Gamma;\Pi \overset{\textsf{safe}}{\vdash} \mathtt{e}[\mathtt{e}_i/\mathtt{x}_i][\mathtt{new\ c()/this}] : \mathtt{c}'$ with $\mathtt{P}\vdash\mathtt{c}' \leq \sigma(\mathtt{t}^b)$,

and we can conclude by applying rule ($\leq$-trans). □

5 IMPLEMENTATION

We have developed a prototype compiler that compiles closed programs using unknown types into standard `.class` files, which can be run on any JVM. We have written a small compiler, instead of trying to modify the standard Java one, to be able to experiment with some examples in a short time; of course, if the full Java language was to be supported, then modifying the standard compiler would be a better choice.

The prototype can be directly tested (using any Java-enabled web browser) at the following web page:

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

The information for obtaining the sources of the prototype can be found at the same URL.

Our compiler applies the ideas described in Section 3 to a larger subset of Java including: constructors, fields, some statements, some primitive types (integers and booleans) and `void` methods. Fortunately, our approach has scaled smoothly to these additional features.

We decided to support constructor overloading, differently from methods. Indeed method overloading can be simulated/avoided by renaming, whereas this solution would have been awkward for constructors. However, for the sake of simplicity, we forbid the use of unknown type parameters in constructors. We may consider to extend overloading resolution to unknown types in future versions.

Fields must be declared of a known type and their accesses are handled like method invocations are: through reflection when unknown (target) types are involved and with standard field accesses everywhere else. The only major difference is that we need to distinguish between read and write accesses invoking, respectively, a (reflective) getter or a (reflective) setter method. Supporting fields of unknown types is a very challenging issue to be investigated in the future.

Including primitive types, though straightforward, required to take care of the fact that unknown types are represented by the standard type `Object` in the translation. That is, values of primitive types need to be boxed and unboxed when they are, respectively, passed-to or returned-from a polymorphic method. Although our prototype generates a Java source that is compiled invoking `javac`, the standard Java compiler, we need to deal with this issue anyway because autoboxing conversions of Java 5 cannot implicit convert from the type `Object` to primitive types.

Our running example, along with its translation, generated by our prototype compiler, can be found in the Appendix.

6 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, even considering Java generics, 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, in contrast to Java generics that provide quantification over types described by explicitly named bounds.

Ideally unknown types and Java generics should be independent, not competing, approaches that can be freely mixed and matched. In this ideal view, programmers would use our "?" annotations where they do not want to fix the types of parameters, regardless the involved constructor or method is itself generic or declared inside a generic type. Alas, in practice this view falls short because the inferred type constraints need to take generic types into considerations. This could make the approach much more complex, so studying the interactions between our approach and standard generics is the subject of further work.

We achieved our goal mixing known technologies like reflection and inferred type constraints for Java-like languages [1] in a novel way. As required for true applicability, there is zero runtime overhead on code which does not take advantage of the new features.

An alternative implementation technique[9] could be an heterogenous translation

[9]Suggested by an anonymous referee of [9].

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 argument types 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. In Java this means that binary compatibility is not preserved. 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 [2]. Previous work on inferring type constraints for object oriented languages includes [12, 5, 4, 11, 13].

Our type constraints are somewhat reminiscent of *where-clauses* [3, 10] 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 argument 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.

Union types [6] are an interesting approach to handle objects of different types via their common interface in a Java-like language. Using union types programmers can exploit the commonalities among different types without any need to modify them. In other words, these types are not required to implement an explicitly named interface, as it would be the case in standard Java. On the other hand, the common interface is sometimes not enough: the common interface of our example classes `List` and `Archive` contains only the method `size` (returning the union type `Integer` $\vee$ `Long`). The method `append` is missing in the union type `List` $\vee$ `Archive` since the two versions have different parameter types. This fact prevents a method like our example `appAndSize` to be typechecked.

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, overloading and overriding of polymorphic methods are important features which are to be taken into account in order to obtain a practical extension of Java.

Acknowledgments We thank OOPS'06 and JOT referees, and Davide Ancona for their useful comments. This work has been partially supported by APPSEM II - Thematic network IST-2001-38957, and MIUR EOS - Extensible Object Systems.

REFERENCES

[1] Davide Ancona, Ferruccio Damiani, Sophia Drossopoulou, and Elena Zucca. Polymorphic bytecode: Compositional compilation for Java-like languages. In *ACM Symp. on Principles of Programming Languages 2005*. ACM Press, January 2005.

[2] Davide Ancona, Giovanni Lagorio, and Elena Zucca. Type inference for polymorphic methods in Java-like languages. Technical report, Dipartimento di Informatica e Scienze dell'Informazione, Università di Genova, October 2006.

[3] Mark Day, Robert Gruber, Barbara Liskov, and Andrew C. Myers. Subtypes vs. where clauses: Constraining parametric polymorphism. In *ACM Symp. on Object-Oriented Programming: Systems, Languages and Applications 1995*, volume 30(10) of *SIGPLAN Notices*, pages 156–168, 1995.

[4] Jonathan Eifrig, Scott F. Smith, and Valery Trifonov. Sound polymorphic type inference for objects. In *ACM Symp. on Object-Oriented Programming: Systems, Languages and Applications 1995*, volume 30(10) of *SIGPLAN Notices*, pages 169–184, 1995.

[5] Jonathan Eifrig, Scott F. Smith, and Valery Trifonov. Type inference for recursively constrained types and its application to OOP. In *Mathematical Foundations of Programming Semantics*, volume 1 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 1995.

[6] Atsushi Igarashi and Hideshi Nagira. Union types for object-oriented programming. In Hisham Haddad, editor, *Proceedings of the 2006 ACM Symposium on Applied Computing (SAC), Dijon, France, April 23-27, 2006*, pages 1435–1441. ACM, 2006.

[7] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *ACM Symp. on Object-Oriented Programming: Systems, Languages and Applications 1999*, pages 132–146, November 1999.

[8] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.

[9] Giovanni Lagorio and Elena Zucca. Introducing safe unknown types in Java-like languages. In L.M. Liebrock, editor, *ACM Symp. on Applied Computing (SAC 2006), Special Track on Object-Oriented Programming Languages and Systems*, pages 1429–1434. ACM Press, 2006.

[10] Andrew C. Myers, Joseph A. Bank, and Barbara Liskov. Parameterized types for Java. In *ACM Symp. on Principles of Programming Languages 1997*, pages 132–145. ACM Press, 1997.

[11] Jens Palsberg. Type inference for objects. *ACM Comput. Surv.*, 28(2):358–359, 1996.

[12] Jens Palsberg and Michael I. Schwartzbach. Object-oriented type inference. In *ACM Symp. on Object-Oriented Programming: Systems, Languages and Applications 1991*, pages 146–161, 1991.

[13] Tiejun Wang and Scott F. Smith. Precise constraint-based type inference for Java. In *ECOOP'01 - European Conference on Object-Oriented Programming*, volume 2072, pages 99–117. Springer, 2001.

A APPENDIX

This appendix contains the running example of the paper and its translation, generated by our prototype compiler.

Source code

```
class Integer {}

class Long {}

class List {
    void append(List l) {
        /* ... */
    }
    Integer size() {
        return new Integer();
    }
}

class Archive {
    void append(Archive a) {
        /* ... */
    }
    Long size() {
        return new Long();
    }
}

class Test {
    void appAndSize(x, y) {
        x.append(y);
        System.out.println(x.size());
    }
    void main() {
        List l1 = new List();
        List l2 = new List();
        Archive a1 = new Archive();
        Archive a2 = new Archive();
        appAndSize(l1, l2);
        appAndSize(a1, a2);
    }
}
```

Translated code

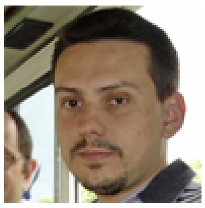
```
class Integer extends Object { public Integer () { } }
class Long extends Object { public Long () { } }

class List extends Object {
    public List () { }
    public void append (List l) { }
    public Integer size () {
        return new Integer() ;
    }
}

class Archive extends Object {
    public Archive () { }
    public void append (Archive a) { }
    public Long size () {
        return new Long() ;
    }
}

class Test extends Object {
    public void appAndSize (Object /* 'a */ x, Object /* 'b */ y) {
        RuntimeLibrary.reflectiveInvoke(x,"append",y) ;
        System.out.println(""+(RuntimeLibrary.reflectiveInvoke(x,"size"))+"") ;
    } // constraints:
      // !void 'e
      // method('a.size(<>)=<> returns 'e)
      // method('a.append(<'b>)=<'c> returns 'd)
    public static void main(String [] args) { new Test().main() ; }
                                 // trampoline method for program startup
    public Test () { }
    public void main () {
        List l1 = new List() ;
        List l2 = new List() ;
        Archive a1 = new Archive() ;
        Archive a2 = new Archive() ;
        /* in: ['a->List, 'b->List] */
        /* out: ['c->List, 'e->Integer, 'd->void] */
            this.appAndSize(l1,l2) ;
        /* in: ['a->Archive, 'b->Archive] */
        /* out: ['c->Archive, 'e->Long, 'd->void] */
            this.appAndSize(a1,a2) ;
    }
}
```

ABOUT THE AUTHORS



Giovanni Lagorio took a Ph.D. in Computer Science at the University of Genova on May 2004. His research interests are in the area of programming languages; in particular, design and foundations of modular and object-oriented languages and systems. He can be reached at lagorio@disi.unige.it.
See also http://www.disi.unige.it/person/LagorioG/.

Elena Zucca Associate professor at the University of Genova since 1999, previously assistant professor at the University of Genova since 1989. Author of more than 40 papers in international journals and conferences. Her main research contributions are in the semantics and specification of concurrent and object-oriented languages, extension of algebraic techniques to dynamic systems, module calculi, type systems and semantics of Java-like languages. She can be reached at zucca@disi.unige.it.
See also http://www.disi.unige.it/person/ZuccaE/.