From FGJ to Java according to LM translator

Mirko Viroli
DEIS – Università di Bologna
via Rasi e Spinelli 176
47023 Cesena (FC), Italy
mviroli@deis.unibo.it

ABSTRACT

In this paper we present a formalization of LM translator [11], a proposal for adding parametric polymorphism to Java by means of a logical extension of Generic Java [6]. LM translator overcomes the type-integration lacks of Generic Java thanks to several distinctive features: run-time information on parametric types is carried into type descriptors created at load-time, reflective features of Java are exploited for dealing with legacy classes, hashtables are used to tackle performance issues, and special method descriptors tables support dynamic dispatching of method calls. Because of all these tricks LM translator turns out to be a complex system, whose description and understanding are often quite complicated, whereas translation approaches were typically used also because of their simplicity.

So, the declared goal of the formalization is to help reasoning about LM, reaching a good trade-off between compactness of its description and completeness in treating its features. Technically, this is done by modeling LM translator as a compilation of FGJ [3], a core calculus for Generic Java, into full Java. This choice for the source and target languages allows to directly focus on the key issues related to parametric polymorphism, thanks to the minimality of FGJ, and not to constrain in any way the power and features of LM translator, as the target language is full Java. The formalization obtained is satisfactorily compact; the only significant issues left out are the management of legacy classes and the internal details of the library classes supporting the translation.

In this paper we also argue that this kind of formalization is the most suitable support for the development of an actual prototype of the translator. This is motivated by outlining our current work on this direction, in which the formalization here introduced plays a crucial role. The key property of our methodology is the separation between the peculiar aspects of the translation and the details not strictly related to parametric polymorphism. This is meant to speed up the tuning of the translation towards an optimized implementation for the Java programming language.

1. INTRODUCTION AND MOTIVATION

In the context of the proposals for adding generics to the Java programming language as a response to Sun's call [1], Generic Java (GJ) [6] is going to be the basis for the first actual release of Java providing this extension [8]. Its basic idea is to fully rely on the so-called *erasure technique*, that is translating a parametric class into the Java class one would have written without having parametric polymorphism and exploiting the homogeneous generic idiom [5]. All the information on the type parameters of a given parametric class or method is completely lost in the translated code, as parametric classes and methods are translated into their monomorphic version, e.g. `List<String>` is translated to `List`, and each type variable is translated to its bound (`Object` by default). This technique allows GJ to enjoy full upward compatibility properties and also avoids almost any translation overhead [6].

However, GJ has also a well recognized limitation. Due to type erasure, those operations involving run-time inspection of the type of an object, basically type-casts and instance tests (Java operator `instanceof`), cannot be supported for parametric types. Being unable to do this may be a serious constraint. On the one hand, even though one adds parametric polymorphism to the language, the need for using heterogeneous collections of elements remains, therefore, the problem of accessing objects previously up-casted still exists. Type-casts and instance tests are the only way to safely recover the proper type of such objects. On the other hand, these type-dependent operations are the basis for supporting successful Java mechanisms related to persistence such as Java Serialization, Java Remote Method Invocation and JavaBeans. Adding parametric polymorphism to the language but then disabling its integration with these important mechanisms clearly leads to a somewhat incomplete extension. As discussed in [8], however, there is some lack of experience with constructs supporting generic types at run-time, and it is unclear whether the current proposals supporting this mechanism addresses performance and compatibility in the proper way. So, the choice was to rely on GJ anyway, at least until future studies may lead to solutions effectively supporting parametric types at run-time.

Currently, the proposal that seems the most reasonable starting point for this research is LM translator [11]. It extends the behaviour of GJ translator so as to make the code produced carrying the necessary information about the instantiation of the type parameters, into Java objects called *type descriptors* and *method descriptors*. These descriptors are created avoiding unnecessary space overhead and run-time overhead, and are exploited to implement those operations requiring run-time information on the parametric type of an object.

An implementation based on LM translator is likely to be a good compromise between the performance overhead introduced and the expressiveness power gained by the language. However, the description and the understanding of the effects of LM translator on the code are not as simple as for GJ. In fact, LM translator puts together a lot of heterogeneous ingredients. First of all, the type and method descriptors supporting generics at run-time are entirely created at load-time, and Java Reflection is used to integrate this management with legacy classes. Then, special data structures ex-

ploiting hashtables are automatically built by the translator in order to reduce the performance overhead of accessing descriptors. Finally, dynamic dispatching of method calls is supported through the simulation of the management of Virtual Method Tables, exploiting structures of method descriptors called Virtual Parametric Methods Tables [10].

In this situation, a formalization would be a necessary tool for harnessing this complexity. It may allow to focus on the relevant issues abstracting away from unimportant ones, and to describe in a comprehensive way the important ideas so as to avoid risks of incomplete specifications. Among all these aspects, the formalization is usually the key tool for leading to robust implementations.

In principle, we may follow the approach of [3], where the semantics of GJ is given in terms of a translation from a source core language, which is a subset of the language accepted by GJ, to a target core language, which is a subset of Java. In particular, these core languages are called, respectively, Featherweight Generic Java (FGJ) and Featherweight Java (FJ), and they include only those few language features directly involved in the translation of generics. This approach has a number of flavors: (i) it keeps the specification as compact as possible, (ii) it completely defines the features of interest discarding secondary ones, (iii) it allows to capture the core concepts of the translation, and (iv) it allows for proving important correctness results, such as type-preservation. However, all these properties hold together because the ideas behind the corresponding system are few and relatively simple.

For LM translator this would not be possible. The basic problem in fact is to isolate as target language a subset of Java having all the features needed to describe the effects of LM translator on the code. Such a language would not be as small as one would like it to be. Suppose we want to proceed by using as target language an extension of FJ. We should add to FJ (i) reflection, in order to deal with legacy Java classes and to support the management of descriptors, (ii) side-effects and class loading, to model the fundamental idea that descriptors are not created each time by need, but once and for all at the application boot-strap, (iii) static fields of client classes, containing the descriptors they use, and (iv) static methods of parametric classes, storing important facilities for creating their descriptors. Then, it must be also considered that an important role is played by some library class of Java, such as arrays, vectors and hashtables. With such a complex target language it would be basically impossible to prove properties in a paper of reasonable space.

Clearly, one may think of dropping some of the features of LM trying to focus on just few issues, so as to rely on a smaller target language. For instance, we may discard the idea of load-time creation of descriptors, and dropping class-loading and side-effects from the target language. However, doing this will made our description of LM lacking one of the main reasons that motivated its introduction. In general, dropping relevant features simplifies the proof of properties but leads to incomplete specifications.

So, in this paper our goal is to define a compact formalization allowing for a complete description of the features of LM translation. This is done by means of a translation from the source language FGJ (with some minimal extension), which is a subset of the language accepted by LM, to a large language, which is Java itself. This choice will turn out to be the best one to our end, as the complexity of LM translator is mostly harnessed in a couple of pages of rules.

Then, we describe our development of an actual implementation of LM translator, highlighting that the formalization here introduced plays a crucial role in supporting it. Basically, it allows to directly write a core prototype of the system containing all the relevant features of LM translator. This prototype can be gradually turned into the final implementation by just adding the programming constructs left out by FGJ, without the need of dealing with further issues strictly related to parametric polymorphism. Our goal is not to produce a fully-featured implementation for Java, but to have a tool allowing fast prototyping and tuning of LM.

The remainder of the paper is as follows. Section 2 overviews LM translator, and Section 3 depicts FGJ focusing on the notation and on the functions introduced in [3] which will be helpful to our formalization. Section 4 provides the actual compilation of FGJ into Java, and Section 5 a comprehensive example, which helps understanding the key concepts of the formalization. Section 6 discusses how we meant to fill the gap between this formalization and the actual implementation of the translator.

2. LM TRANSLATOR

In the code produced by LM translator [11], each class that needs to perform some operation on parametric types, such as type-casts, instance tests, and object allocations, will handle type descriptors for these types, which are objects of the library class `$TD`. In particular, these type descriptors are stored into static fields created by the translator, and are initialized at load-time. When allocating an object from a parametric type (`new` expression), the corresponding descriptor is passed as first argument in the constructor, and will then be used to make that object keeping information on its type. Then, these descriptors are exploited to implement type-casts and instance tests, using the methods `cast` and `isInstance` of the class of descriptors `$TD`. See the signature of class `$TD` in Figure 1.

Similar management is supported for parametric methods, with analogous method descriptors of class `$MD` which are passed at invocation-time. However, here the problem is complicated due to the need of dealing with dynamic dispatching. In [10] we proposed a solution exploiting a data structure called Virtual Parametric Method Table (VPMT). Each descriptor carries its own VPMT, which analogously to a Virtual Methods Table (VMT) [4], allows the proper method descriptor to be bound to the body at invocation-time. The VPMT is an array with one entry for each virtual method (i.e., a public or protected method of Java); each entry contains a vector of method descriptors. The position of a given instantiation of a parametric method is invariant through VPMTs of different subclasses, so by just using this position the actual receiver of the invocation can access the right descriptor [10].

Another key aspect of LM translator is that descriptors are kept by descriptors managers living at run-time, respectively in the class `$TDM` (Type Descriptors Manager) and `$MDM` (Method Descriptors Manager). Their basic goal is to prevent descriptors from being created twice. This is obtained by registering descriptors in their manager each time they are created. Finally, LM translator also relies on a special technique for treating those parametric types that exploit type variables of the scope within their parameters [11]. As the actual instantiation of these types is unknown until the type variables get actually instantiated, we let each type descriptor and method descriptor carrying those type/method descriptors using its parameters, called friend types/methods. Then, when we register a

```
public class $TD {

  public Class c;
  public $TD[] params;
  public $TD[] friendTs;
  public int[] friendMs;
  public $TD father;
  public Vector[] VPMT;

  public $TD(Class c, $TD[] p) { ... }

  public boolean checkNew() { ... }

  boolean isInstance(Object o){
    return (o instanceof $Parametric)
      ?(($Parametric)o).getTD().isSubType(this)
      :c.isInstance(o);
  }
  Object cast(Object o){
    if (isInstance(o)) return o;
    throw new ClassCastException(...);
  }
}
```

```
public class $MD {

  public int mID;
  public int lPos;
  public $TD tR;
  public $TD[] params;
  public $TD[] friendTs;
  public int[] friendMs;

  public $MD($TD t, int mID, $TD[] p) { ... }
  public boolean checkNew() { ... }
}

public class $TDM{
  public static $TD register(Class c) { ... }
  public static $TD register(Class c,$TD[] p) { ... }
  public static void initVPMT($TD t){ ... }
}

public class $MDM {
  public static $MD register($TD tR,int mID,$TD[] p) { ... }
  public static void propagate($TD tR,int mID,$TD[] p){ ... }
}
```

Figure 1: Some detail on the structure of the library classes

descriptor we are able to automatically register its friend descriptors. This is the main task accomplished by the methods `createTD` and `createMD`, built by the translator in each parametric class. See some detail on the implementation of the library classes supporting the translation in Figure 1.

In general, we will assume that the source language for LM translator is the same as that of GJ, even though at this point of our research we haven't faced yet the implementation of issues such as inner parametric classes, parametric exceptions, and so on, which are all managed by GJ. However, we are confident they can be implemented in a satisfactory way in our framework.

Some of the core ideas of LM translator have been borrowed from NextGen, a code-expansion technique for translating generics in Java proposed in [2]. There, type-passing style is exploited for implementing parametric methods, as objects called *snippet* environments are passed at invocation-time in an analogous way of our method descriptors. These environments are created at load-time as well, leading to very small run-time overhead. However, the code-expansion technique leads to high memory and code footprint, so the approaches exploiting this technique will not be used for the actual implementation of the Java programming language, as motivated in [8].

The general idea of LM translator seems not to be strictly related to Java, but of a somewhat general appealing. In fact, the proposal for extending Microsoft's .NET Common Language Runtime with generics shown in [9] uses a similar technique. The main difference is that in this proposal the necessary information on the parametric types is not built eagerly, at load-time, but only once on a by-need basis. This style can be used in LM as well. Basically it implies that type descriptors are not created in the initialization code of the client classes, but the first time they are accessed. We are currently working on this variation of LM, which we believe may lead to a better implementation.

3. FEATHERWEIGHT GENERIC JAVA

The source language for our formalization is Featherweight Generic Java (FGJ) [3], a core-calculus for GJ focusing on parametric classes, parametric methods and fields, and in which expressions can be only type-casts, object allocations, field accessing and method invocations. Language constructs remaining out from FGJ, and from the formalization we are going to introduce as well, are interfaces, inner classes and arrays.

To the end of introducing the syntax of FGJ, we let the metavariables `C` and `D` range over class names; `S`, `T`, and `U` range over types; `X` and `Y` range over type variables; `N` and `P` range over non-variable types; `CL` ranges over class declarations, `K` over constructors, `M` over method declarations, `f` over field names, `m` over method names, `x` over variables, and `e` over expressions. We denote by $\overline{\mathtt{a}}$ a list of elements $\mathtt{a}_1, \mathtt{a}_2, \ldots$, and the empty list by ϵ. Then, we abuse the notation of function application, as a function $f(\mathtt{a}) = \mathtt{b}$ can be used in $f(\overline{\mathtt{a}}) = \overline{\mathtt{b}}$ to denote the application on all the elements of the list $\overline{\mathtt{a}}$, returning the list of results $\overline{\mathtt{b}}$. Substituting the sub-term `a` by `b` into the term `c` is denoted by $[\mathtt{b}/\mathtt{a}]\mathtt{c}$, while $[\overline{\mathtt{a}} \mapsto \overline{\mathtt{b}}]f$ is the function mapping $\overline{\mathtt{a}}$ to $\overline{\mathtt{b}}$ and any other element `c` to $f(\mathtt{c})$. The grammar of FGJ is the following:

```
CL  ::=  class C<X ◁ N> ◁ N{T f; K M}
K   ::=  C(T f){super(f); this.f=f;}
M   ::=  <X ◁ N> T m (T x){↑ e;}
e   ::=  x | e.f | e.m<T>(e)
         | new N(e) | (T)e
T   ::=  X | N
N   ::=  C<T>
```

The environments Δ and Γ are used to model the type system of FGJ. $\Delta = \overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}$ maps type variables into their bounds, which are non-variable types. $\Gamma = \overline{\mathtt{y}} : \overline{\mathtt{T}}$ maps variables into types, that is formal parameters to their declared type. Then, we have the following judgements (see [3] for their actual semantics):

- The subtyping judgement $\Delta \vdash \mathtt{T} <: \mathtt{U}$, stating that the type T is a subtype of U under the environment Δ.
- Expression typing judgement $\Delta; \Gamma \vdash \mathtt{e} \in \mathtt{T}$, that under a Δ environment and a Γ environment, gives the type T to the expression e.
- Class typing judgement CL OK, stating whether a class definition is well-formed and well-typed.

We suppose the existence of a class table CT, taking class names and returning class definitions. Among other usual properties, we suppose that the root type Object is in the dominion of CT, and that all the class definitions are OK.

Our version of FGJ is actually slightly different from the one presented in [3]. In order to stress the fact that LM translator allows for implementing type-operations without any limitation we changed the syntax of type-casts, allowing to cast an object not only to a non-variable type N as in [3], but to a parametric type T in general. The corresponding change on the type system and on the relative proofs should be minimal. In particular, now we don't need any rule for checking valid down-casts as in [3], as all type-casts are compilable by LM translation. To the end of defining our translation we need the following look-up functions defined in [3]:

- *mtypemax*$(\mathtt{m}, \mathtt{C}) = \overline{\mathtt{D}} \mapsto \mathtt{D}$, returning the erased argument types $\overline{\mathtt{D}}$ and return type D of the method m in C. In particular, this is done by finding the method in the highest superclass in which it is defined.
- *fieldsmax*$(\mathtt{C}) = \overline{\mathtt{D}}\ \overline{\mathtt{f}}$, returning couples - erased type, field name - of the class C, finding field types in the highest superclass in which they are defined.

Looking for members in the highest class and comparing the result with standard lookup functions permits to determine whether some type variable has been instantiated due to an extend clause. In such a case in fact, the so-called *stupid casts* should be automatically inserted by the translator [3].

4. THE FORMALIZATION

This section presents the core of the paper, that is the compilation of FGJ into Java according to the translation schema of LM introduced in [11, 10]. As already mentioned our version of FGJ provides full type-cast ability. [11] highlights that translating type-casts is much the same than translating instance tests. So, the translation of type-casts is a mean for the translation of type-dependent operations in general.

This compilation abstracts away from the actual translation of non-parametric classes and methods. For instance, in an actual implementation the classes that do not have type parameters will not keep a local type descriptor, and methods without type parameters will not need to receive the method descriptor as first argument, and so on. In general the final translator should be able to translate pure Java sources to themselves. Then, here we also abstract from the details on the implementation of library classes $TD, $MD, $TDM and $MDM. An interested reader can refer to [11, 10].

The whole translation resembles the one shown in [3], both on notation and on semantics. The two basic differences are that LM translator provides special translation for operations involving parametric types and/or instantiating type parameters, and that the proper code to create descriptors and to keep track of them should be added to each class.

4.1 Auxiliary functions

The basic idea behind LM translator is to handle Java objects called type descriptors and methods descriptors, containing the necessary information to translate casts, object allocations and method invocations, included in the corresponding class or method, respectively. While a type descriptor is completely identified by the corresponding non variable type N, for method descriptors we use *method signatures* L, where $\mathtt{L} ::= \mathtt{T.m<}\overline{\mathtt{T}}\mathtt{>}$.

The information on what type descriptors and method descriptors have to be created is statically gathered at translation-time, and are modeled by means of the functions *getT* and *getM*. In particular, *getT* is used to get the parametric types used in casts, allocations and as receivers of method calls. This can be done either (i) from an expression e in the environments Δ and Γ (*getT*$_{\Delta,\Gamma}(\mathtt{e})$), (ii) from the body of a method M defined in class C (*getT*$(\mathtt{C}, \mathtt{M})$) or (iii) from all the expressions contained in class C (*getT*$(\mathtt{C})$). Analogously, *getM* gets the signature of the parametric methods invoked (i) from an expression e in the environments Δ and Γ (*getM*$_{\Delta,\Gamma}(\mathtt{e})$), (ii) from the body of a method M defined in class C (*getM*$(\mathtt{C}, \mathtt{M})$) and (iii) from all the expressions contained in a class C (*getM*$(\mathtt{C})$). Their semantics is shown in Figure 2. The operator $\bullet$ is meant to join two lists or an element and a list, discarding duplicates and preserving order. For instance we have $\mathtt{a,b,c} \bullet \mathtt{a,d,c,f} = \mathtt{a,b,c,d,f}$. Basically, the function $getT$ gathers the types used in casts, allocations and as receivers of method calls, while the function $getM$ gathers the signatures of the methods invoked.

We divide parametric types and method signatures into those having fully-instantiated type parameters, called *bound types* and *bound methods*, and those that instead contain some type variable, called *free types* and *free methods*, respectively. In the former case their descriptors are completely known, so they can be registered at the load-time of the classes which use them. In the latter case, instead, they are registered only when the descriptor of the enclosing class/method is registered, as only at this time the instantiation of the type parameters is known. We introduce the predicates *boundT*$_\Delta$(N), *freeT*$_\Delta$(N), *boundM*$_\Delta$(L) and *freeM*$_\Delta$(L) to check if under the environment Δ, specifying the type variables of the scope, the type N and the signature L are, respectively, bound or free. The notation for these predicates will be abused, so that when applying them to a list the result is the sublist of the elements satisfying the predicate. We have:

$$
\begin{array}{lcl}
\textit{freeT}_\Delta(\mathtt{N}) & \Leftrightarrow & \exists \mathtt{X} \in \textit{dom}(\Delta), \exists \mathtt{O} : \ \mathtt{[O/X]N} \neq \mathtt{N} \\
\textit{freeM}_\Delta(\mathtt{L}) & \Leftrightarrow & \exists \mathtt{X} \in \textit{dom}(\Delta), \exists \mathtt{O} : \ \mathtt{[O/X]L} \neq \mathtt{L} \\
\textit{boundT}_\Delta(\mathtt{N}) & \Leftrightarrow & \text{not} \quad \textit{freeT}_\Delta(\mathtt{L}) \\
\textit{boundM}_\Delta(\mathtt{L}) & \Leftrightarrow & \text{not} \quad \textit{freeM}_\Delta(\mathtt{L})
\end{array}
$$

We provide facilities for accessing the type descriptor of a given type and the method descriptor of a given method signature. We introduce two environments for types and signatures, denoted by symbols Θ and Π, respectively. $\Theta = \overline{\mathtt{T}} \mapsto \overline{\mathtt{e}}_J$ associates types to Java expressions representing the corresponding descriptor, $\Pi_\Theta = \overline{\mathtt{L}} \mapsto \overline{\mathtt{e}}_J$ binds method signatures to Java expressions representing the corresponding descriptors, the latter possibly exploiting the Θ environment to resolve some type descriptor. We access the de-

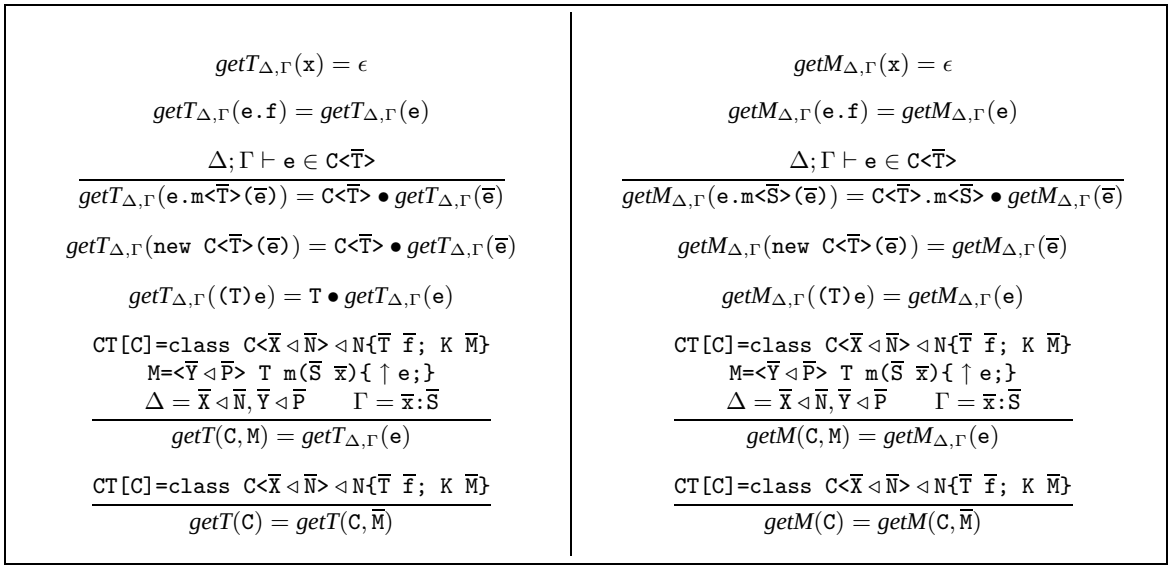

Figure 2: Gathering functions for type and method descriptors

scriptor for a type T by the notation $|\mathtt{T}|_\Theta$ and that of a method signature L by $|\mathtt{L}|_{\Pi,\Theta}$. Then, we introduce a standard Θ environment denoted by Θ_s, implementing the registration of bound types, defined as:

$$\Theta_s = \mathtt{C<}\overline{\mathtt{T}}\mathtt{>} \mapsto \mathtt{C.createTD(new\ \$TD[]\{}|\overline{\mathtt{T}}|_{\Theta_s}\mathtt{\})}$$

that is, exploiting the static method createTD of the class, which accepts the array of type descriptors for the parameters. When a Θ environment has to deal with free types as well, we should add the specification on how to resolve type variables, and this can be done exploiting the environment $[\overline{\mathtt{X}} \mapsto \overline{\mathtt{e}}_J]\Theta$, according to our notation for function substitution. Then, we also introduce a standard Π environment Π_s as follows:

$$\frac{\begin{array}{c}\mathtt{CT[C]=class\ C<}\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\mathtt{>} \lhd \mathtt{N\{}\overline{\mathtt{T}}\ \overline{\mathtt{f}}\mathtt{;\ K\ }\overline{\mathtt{M}}\mathtt{\}} \\ \mathtt{M_i=<}\overline{\mathtt{Y}} \lhd \overline{\mathtt{P}}\mathtt{>\ T\ m(}\overline{\mathtt{S}}\ \overline{\mathtt{x}}\mathtt{)\{}\uparrow \mathtt{e;\}} \qquad \mathit{pos}(\mathtt{C},\mathtt{M}_i) = j\end{array}}{\begin{array}{l}\Pi_{s\Theta} = \mathtt{C<}\overline{\mathtt{T}}\mathtt{>.m<}\overline{\mathtt{U}}\mathtt{>} \mapsto \\ \qquad \mathtt{C.createMD(}|\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}|_\Theta, j, \mathtt{new\ \$TD[]\{}|\overline{\mathtt{U}}|_\Theta\mathtt{\})}\end{array}}$$

The static method createMD is created by the translator, and accepts the type descriptor of the receiver, the position of m in C, and an array containing the descriptors for the parameters. The semantics of function *pos*(C, M) is described in Figure 3. It returns the position of M in the VPMT of C. Basically, a method adds a new element to the VPMT only if it does not override a method in the super-class, otherwise its position is *inherited* from that method. The content of the methods createTD and createMD will be shown in the next sections.

4.2 Erasing types

The erasure of types is mostly the same as that of FGJ. We have the function returning the bound of a type as:

$$\mathit{bound}_\Delta(\mathtt{X}) := \Delta(\mathtt{X}) \qquad \mathit{bound}_\Delta(\mathtt{N}) := \mathtt{N}$$

and then the erasure function from FGJ types to Java classes:

$$|\mathtt{T}|_\Delta := \mathtt{C} \quad \text{if} \quad \mathit{bound}_\Delta(\mathtt{T}) = \mathtt{C<}\overline{\mathtt{T}}\mathtt{>}, \quad \mathtt{T} \neq \mathtt{Object<>}$$

$$|\mathtt{Object<>}|_\Delta := \mathtt{LMObj}$$

In this formalization Object is translated into a special library class LMObj, as shown in top of Figure 3.

4.3 Translating Classes

For the rules defining translation of classes refer to Figure 3. FGJ classes are erased to Java classes: (i) by providing the proper translation of fields (erasing their types), methods ($|\mathtt{M}|_\mathtt{C}$) and constructor ($|\mathtt{K}|$), (ii) by adding the static methods createTD and createMD handling the creation of free types and methods (obtained by $|\mathtt{C}|_{CTD}$ and $|\mathtt{C}|_{CMD}$, respectively), and (iii) by adding static fields meant to contain bound descriptors, according to the functions *buildSTD*() and *buildSMD*(). The translation of the root class Object is similar, but we don't have methods, createMD and static fields.

4.4 Static Type and Method Descriptors

Bound type descriptors and method descriptors are completely known at compile-time, so they can be created once and for all at the class load-time, i.e., in the initialization code of newly-created static fields, exploiting standard environments Θ_s and Π_s. In particular, when registering a parametric type the actual descriptor is yielded, while registering a method returns the position of the method descriptor in the VPMT (for details on this, see [10]). Then, because of the special management of VPMTs, instead of creating the descriptors for the free method signature L, we create its version $\mathit{topM}_\Delta(\mathtt{L})$ in the highest super-class defining it. In fact, registering a method descriptors will cause a down-propagation of registrations on all the sub-types, so starting from the top version guarantees all the VPMTs to be properly completed.[1]

4.5 Translating the Constructor

The translation of a constructor K provides for the extra-argument, containing the type descriptor of the current instance. The field

[1]These are details of the library classes supporting the translation, which are mostly unimportant here.

Translation for classes:

$$
\begin{array}{l}
|\texttt{class Object<> extends Object<>\{\}}| = \\
\lceil\ \texttt{class LMObj extends Object \{} \\
\quad\ \texttt{\$TD td; \$TD getTD()\{return td;\}} \\
\quad\ \texttt{LMObj(\$TD td)\{this.td=td;\}} \\
\quad\ |\texttt{C}|_{CTD} \\
\lfloor\ \texttt{\}}
\end{array}
$$

$$
\frac{\begin{array}{c}\Delta = \overline{\texttt{X}} \triangleleft \overline{\texttt{N}} \qquad \mathit{buildSTD}(\texttt{C}) = \overline{\texttt{f}}_{ST}\ \ \overline{\texttt{e}}_{ST} \\ \mathit{buildSMD}(\texttt{C}) = \overline{\texttt{f}}_{SM}\ \ \overline{\texttt{e}}_{SM} \qquad \texttt{C} \neq \texttt{Object}\end{array}}{\begin{array}{l}|\texttt{class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{> } \triangleleft \texttt{N\{}\overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{; K }\overline{\texttt{M}}\texttt{\}}| = \\ \lceil\ \texttt{class C extends } |\texttt{N}|_\Delta\ \texttt{\{} \\ \quad\ \texttt{static } \overline{\texttt{TD}}\ \overline{\texttt{f}}_{ST}\texttt{=}\overline{\texttt{e}}_{ST}\texttt{;} \\ \quad\ \texttt{static int } \overline{\texttt{f}}_{SM}\texttt{=}\overline{\texttt{e}}_{SM}\texttt{;} \\ \quad\ |\overline{\texttt{T}}|_\Delta\ \overline{\texttt{f}}\texttt{;}\ \ |\texttt{C}|_{CTD}\ \ |\texttt{C}|_{CMD}\ \ |\texttt{K}|\ \ |\overline{\texttt{M}}|_{\texttt{C}} \\ \lfloor\ \texttt{\}}\end{array}}
$$

Static descriptors initilization:

$$
\frac{\begin{array}{c}\texttt{CT[C]=class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{> } \triangleleft \texttt{N\{...\}} \\ \Delta = \overline{\texttt{X}} \triangleleft \overline{\texttt{N}} \qquad \overline{\texttt{N}}_B\texttt{=}\mathit{boundT}_\Delta(\mathit{getT}(\texttt{C}))\end{array}}{\mathit{buildSTD}(\texttt{C}) = \overline{\texttt{f}}_{ST} \quad |\overline{\texttt{N}}_B|_{\Theta_s}}
$$

$$
\frac{\begin{array}{c}\texttt{CT[C]=class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{> } \triangleleft \texttt{N } \texttt{\{...\}} \\ \Delta = \overline{\texttt{X}} \triangleleft \overline{\texttt{N}} \qquad \overline{\texttt{L}}_B\texttt{=}\mathit{topM}_\Delta(\mathit{boundM}_\Delta(\mathit{getM}(\texttt{C})))\end{array}}{\mathit{buildSMD}(\texttt{C}) = \overline{\texttt{f}}_{SM} \quad |\overline{\texttt{L}}_B|_{\Pi_s,\Theta_s}}
$$

Translating the constructor:

$$
\frac{\begin{array}{c}\texttt{CT[C]=class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{> } \triangleleft \texttt{N\{}\overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{;K }\overline{\texttt{M}}\texttt{\}} \\ \Delta = \overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\end{array}}{\begin{array}{l}|\texttt{C(}\overline{\texttt{T}}'\ \overline{\texttt{f}}'\texttt{,}\overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{)\{...\}}| = \\ \lceil\ \texttt{\$TD td; \$TD getTD()\{return td;\}} \\ \texttt{C(\$TD td,}|\overline{\texttt{T}}'|_\Delta\ \overline{\texttt{f}}'\texttt{, }|\overline{\texttt{T}}|_\Delta\ \overline{\texttt{f}}\texttt{)\{} \\ \quad\ \texttt{super(td.father,}\overline{\texttt{f}}'\texttt{);} \\ \quad\ \texttt{this.td=td;this.}\overline{\texttt{f}}\texttt{=}\overline{\texttt{f}}\texttt{;} \\ \lfloor\ \texttt{\}}\end{array}}
$$

Implementing `createTD`**:**

$$
\frac{\begin{array}{c}\texttt{CT[C]=class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{> } \triangleleft \texttt{N\{}\overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{; K }\overline{\texttt{M}}\texttt{\}} \\ \overline{\texttt{N}}_F\texttt{=}\mathit{freeT}_\Delta(\mathit{getT}(\texttt{C})) \qquad \Theta_p = [\texttt{X}_i \mapsto \texttt{p[i]}]\Theta_s \\ \overline{\texttt{L}}_F\texttt{=}\mathit{topM}_\Delta(\mathit{freeM}_\Delta(\mathit{getM}(\texttt{C}))) \qquad \Delta\texttt{=}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\end{array}}{\begin{array}{l}|\texttt{C}|_{CTD} = \\ \lceil\ \texttt{static \$TD createTD(\$TD[] p)\{} \\ \quad\ \texttt{\$TD t=\$TDM.register(C.class,p);} \\ \quad\ \texttt{if (t.checkNew())\{} \\ \quad\quad\ \texttt{t.father=}|\texttt{N}|_{\Theta_p}\texttt{;} \\ \quad\quad\ \texttt{t.friendTs=new \$TD[]\{}|\overline{\texttt{N}}_F|_{\Theta_p}\texttt{\};} \\ \quad\quad\ \texttt{t.friendMs=new int[]\{}|\overline{\texttt{L}}_F|_{\Pi_s,\Theta_p}\texttt{\};} \\ \quad\quad\ \texttt{t.initVPMT(); \}} \\ \quad\ \texttt{return t;} \\ \lfloor\ \texttt{\}}\end{array}}
$$

Auxiliary functions:

$$
\mathit{meths}(\texttt{Object}) = \epsilon
$$

$$
\frac{\texttt{CT[C]=class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{> } \triangleleft \texttt{D<}\overline{\texttt{U}}\texttt{>\{}\overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{; K }\overline{\texttt{M}}\texttt{\}} \quad \texttt{M}_i\texttt{=<}\overline{\texttt{Y}}_i \triangleleft \overline{\texttt{P}}_i\texttt{> T}_i\ \texttt{m}_i\texttt{(}\overline{\texttt{S}}_i\ \overline{\texttt{x}}_i\texttt{)\{} \uparrow \texttt{e}_i\texttt{;\}}}{\mathit{meths}(\texttt{C}) = \mathit{meths}(\texttt{D}) \bullet \overline{\texttt{m}}}
$$

$$
\frac{\mathit{meths}(\texttt{C}) = \overline{\texttt{m}} \qquad \texttt{M} = \texttt{<}\overline{\texttt{Y}} \triangleleft \overline{\texttt{P}}\texttt{> T m}_i\texttt{(}\overline{\texttt{T}}\ \overline{\texttt{x}}\texttt{)\{} \uparrow \texttt{e;\}}}{\mathit{pos}(\texttt{C},\texttt{M}) = i}
$$

$$
\frac{\begin{array}{c}\texttt{CT[C] = class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{> } \triangleleft \texttt{N\{}\overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{; K }\overline{\texttt{M}}\texttt{\}} \\ \texttt{M}_i\texttt{=<}\overline{\texttt{Y}} \triangleleft \overline{\texttt{P}}\texttt{> T m(}\overline{\texttt{U}}\ \overline{\texttt{x}}\texttt{)\{return e;\}} \qquad \Delta = \overline{\texttt{Y}} \triangleleft \overline{\texttt{P}}\end{array}}{\begin{array}{l}\mathit{pMeth}(\texttt{C},\texttt{M}_i) = \mathit{topM}_\Delta(\mathit{freeM}_\Delta(\mathit{getM}(\texttt{C},\texttt{m}))) \ ; \\ \mathit{freeT}_\Delta(\mathit{getT}(\texttt{C},\texttt{m})) \ ; \ [\texttt{Y}_j \mapsto \texttt{p}[j]][\texttt{X}_k \mapsto \texttt{td.p}[k]]\Theta_s\end{array}}
$$

$$
\frac{\texttt{CT[C] = class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{> } \triangleleft \texttt{N } \texttt{\{}\overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{; K }\overline{\texttt{M}}\texttt{\}} \qquad f : i \mapsto \epsilon; \epsilon; \varnothing}{\mathit{mEnv}(\texttt{C}) = [\mathit{pos}(\texttt{C},\texttt{M}_i) \mapsto \mathit{pMeth}(\texttt{C},\texttt{M}_i)]f}
$$

Implementation of `createMD`**:**

$$
\frac{\begin{array}{c}\texttt{CT[C]=class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{> } \triangleleft \texttt{N } \texttt{\{}\overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{; K }\overline{\texttt{M}}\texttt{\}} \\ \mathit{mEnv}(\texttt{C},\texttt{M}_i) = i \mapsto \overline{\texttt{L}}_i; \overline{\texttt{N}}_i; \Theta_{\texttt{i}}\end{array}}{\begin{array}{l}|\texttt{C}|_{CMD} = \\ \lceil\ \texttt{static int createMD(\$TD t,int pos,\$TD[] p)\{} \\ \quad\ \texttt{\$MD m=\$MDM.register(t,pos,p);} \\ \quad\ \texttt{if (m.checkNew())\{} \\ \quad\quad\ \texttt{t.VPMT[pos].addElement(m);} \\ \quad\quad\ \texttt{m.lPos=t.VPMT[pos].size()-1;} \\ \quad\quad\ \texttt{if (pos==}\tilde{i}\texttt{)\{} \\ \quad\quad\quad\ \texttt{t.friendTs=new \$TD[]\{}|\overline{\texttt{N}}_i|_{\Theta_{\texttt{i}}}\texttt{\};} \\ \quad\quad\quad\ \texttt{t.friendMs=new int[]\{}|\overline{\texttt{L}}_i|_{\Pi_s,\Theta_{\texttt{i}}}\texttt{\};} \\ \quad\quad\ \texttt{\}} \\ \quad\quad\ \texttt{\$MDM.propagate(m); \}} \\ \quad\ \texttt{return m.lPos;} \\ \lfloor\ \texttt{\}}\end{array}}
$$

Environments for a method:

$$
\frac{\begin{array}{c}\texttt{CT[C]=class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{> } \triangleleft \texttt{N\{}\overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{; K }\overline{\texttt{M}}\texttt{\}} \qquad \Delta = \overline{\texttt{X}} \triangleleft \overline{\texttt{N}} \\ \mathit{pMeth}(\texttt{C},\texttt{M}_j) = \overline{\texttt{L}}_M; \overline{\texttt{N}}_M; \Theta_{\texttt{M}} \qquad \mathit{pos}(\texttt{C},\texttt{M}_j) = l \\ \mathit{buildSTD}(\texttt{C}) = \overline{\texttt{f}}_{ST} \ \ |\overline{\texttt{N}}_B|_{\Theta_s}; \qquad \mathit{freeT}_\Delta(\mathit{getT}(\texttt{C})) = \overline{\texttt{N}}_F \\ \texttt{e}_m = \texttt{((\$MD)(td.VPMT[}l\texttt{].elementAt(md)))}\end{array}}{\begin{array}{rcl}\Theta^{\texttt{C},\texttt{M}_j} & = & [\texttt{C<}\overline{\texttt{X}}\texttt{>} \mapsto \texttt{td}]\texttt{N}_{Fi} \mapsto \texttt{td.friendTs[}i\texttt{]}, \\ & & \overline{\texttt{N}}_{Mk} \mapsto \texttt{e}_m\texttt{.friendTs[}k\texttt{]}, \overline{\texttt{N}}_B \mapsto \overline{\texttt{f}}_{ST}\end{array}}
$$

$$
\frac{\begin{array}{c}\texttt{CT[C]=class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{> } \triangleleft \texttt{N\{}\overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{; K }\overline{\texttt{M}}\texttt{\}} \qquad \Delta = \overline{\texttt{X}} \triangleleft \overline{\texttt{N}} \\ \mathit{pMeth}(\texttt{C},\texttt{M}_j) = \overline{\texttt{L}}_M; \overline{\texttt{N}}_M; \Theta_{\texttt{M}} \qquad \mathit{pos}(\texttt{C},\texttt{M}_j) = l \\ \texttt{M}_j\texttt{=<}\overline{\texttt{Y}} \triangleleft \overline{\texttt{P}}\texttt{> T m(}\overline{\texttt{U}}\ \overline{\texttt{x}}\texttt{)\{return e;\}} \\ \mathit{buildSMD}(\texttt{C}) = \overline{\texttt{f}}_{SM} \ \ |\overline{\texttt{L}}_B|_{\Pi_s,\Theta_s}; \\ \mathit{topM}_\Delta(\mathit{freeM}_\Delta(\mathit{getM}(\texttt{C}))) = \overline{\texttt{L}}_F \\ \texttt{e}_m = \texttt{((\$MD)(td.VPMT[}l\texttt{].elementAt(md)))}\end{array}}{\begin{array}{rcl}\Pi^{\texttt{C},\texttt{M}_j} & = & [\texttt{C<}\overline{\texttt{X}}\texttt{>.m<}\overline{\texttt{Y}}\texttt{>} \mapsto \texttt{md}]\texttt{L}_{Fi} \mapsto \texttt{td.friendMs[}i\texttt{]}, \\ & & \overline{\texttt{L}}_{Mk} \mapsto \texttt{e}_m\texttt{.friendMs[}k\texttt{]}, \overline{\texttt{L}}_B \mapsto \overline{\texttt{f}}_{SM}\end{array}}
$$

Translation for methods:

$$
\frac{\begin{array}{c}\texttt{CT[C]=class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{> } \triangleleft \texttt{ N } \texttt{\{}\overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{;K }\overline{\texttt{M}}\texttt{\}} \\ \Delta = \overline{\texttt{X}} \triangleleft \overline{\texttt{N}} \qquad \Gamma = \overline{\texttt{x}}\texttt{:}\overline{\texttt{T}}\texttt{, this:C<}\overline{\texttt{X}}\texttt{>} \\ \mathit{mtypemax}(\texttt{m},\texttt{C}) = \overline{\texttt{D}} \mapsto \texttt{D} \\ \texttt{e}_i = \begin{cases}\texttt{x}'_i & \text{if} \quad \texttt{D}_i = |\texttt{T}_i|_\Delta \\ (|\texttt{T}_i|_\Delta)\texttt{x}'_i & \text{otherwise}\end{cases}\end{array}}{\begin{array}{l}|\texttt{M=T m(}\overline{\texttt{T}}\ \overline{\texttt{x}}\texttt{)\{return e}_0\texttt{;\}}|_{\texttt{C}} = \\ \lceil\ \texttt{D m(int md,}\overline{\texttt{D}}\ \overline{\texttt{x}}'\texttt{)\{} \\ \quad\quad\ \texttt{return [}\overline{\texttt{e}}\texttt{/}\overline{\texttt{x}}\texttt{]}|\texttt{e}_0|_{\Delta,\Gamma,\Theta^{\texttt{C},\texttt{M}},\Pi^{\texttt{C},\texttt{M}}} \\ \lfloor\ \texttt{\}}\end{array}}
$$

Figure 3: Main translation functions

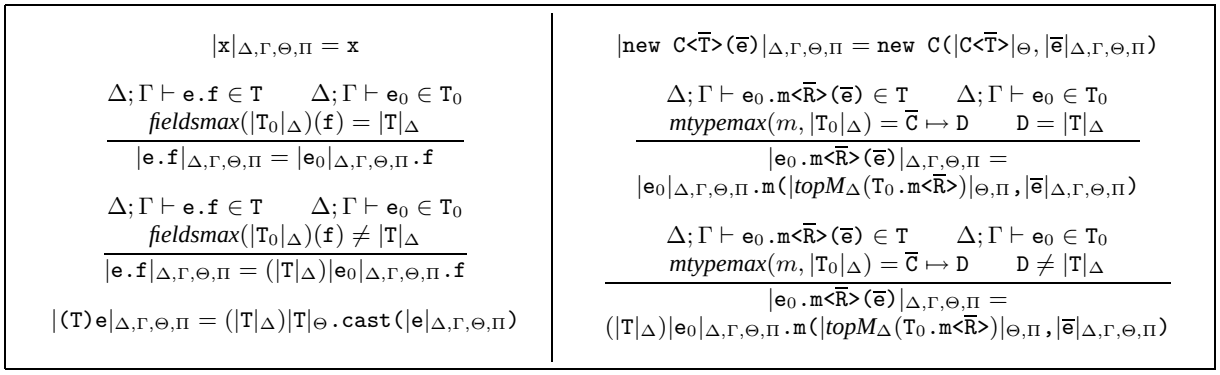

Figure 4: Translating expressions

`$TD.father` is used to pass to the super-class its descriptor. Then, the descriptor is stored into an instance field `td`, inserted by the translation in each class, and yielded by a method `getTD` [2].

4.6 Creating type and method descriptors

The method `createTD` accepts the descriptors of the type parameters, and registers the descriptor of the type into `$TDM`. Then, if this is the first time this was registered (controlled with `checkNew`), in `createTD` we also fill the content of the fields (i) `father`, with the descriptor of the direct super-type `N`, (ii) `friendTs`, with the descriptor for the free types of the class, and (iii) `friendMs` with the descriptor for the free methods of the class. This is supported by the Θ_p environment mapping the type variables to the arguments of the method `createTD`.

For method descriptors, first of all we build an auxiliary function *pMeth*(`C`, `M`) taking a class `C` and a method `M`, and returning a triplet of elements $\overline{\mathtt{L}}; \overline{\mathtt{N}}; \Theta$ containing respectively: (i) the free method signatures in `M` exploiting some of its type variables $\overline{\mathtt{Y}}$, (ii) the free types in `M` exploiting some of its type variables $\overline{\mathtt{Y}}$, and (iii) a Θ environment binding `C`'s type variables $\overline{\mathtt{X}}$ and `M`'s type variables $\overline{\mathtt{Y}}$ to the corresponding descriptors. In particular variables $\overline{\mathtt{Y}}$ are associated to the arguments `p` of `createMD`, while variables $\overline{\mathtt{X}}$ are associated to the field `p` of the descriptor `t`, representing the receiver for the invocation. Then, the function *mEnv*(`C`) maps positions in the VPMT to such triplets, leaving blanks $(\epsilon; \epsilon; \oslash)$ the triplets for those methods which are not redefined in `C`, but are just inherited from the super-class. In fact, the function f, which maps each position to ϵ, is filled only with the triplets for the methods actually defined in `C`.

The method `createMD` accepts the receiver descriptor `t`, the unique identifier `pos` of the method, and the descriptors of the parameters `p`. It registers the method descriptors, and in the case this was the first time, it proceeds by filling the rest of the object `m`. In particular, it adds `m` in the VPMT of `t`, it registers its position in `m.lPos` and then, depending on i, it stores the fields `friendTs` and `friendMs`, by exploiting the result of the auxiliary function *pMeth*(`C`, $\mathtt{M}_i$). The presence of $\overline{i}$ in the `if` statement is meant to model a sequence of `if` statements on all the values assumed by i.

4.7 Translating methods

First of all, we build the $\Theta^{\mathtt{C},\mathtt{M}}$ and $\Pi^{\mathtt{C},\mathtt{M}}$ environments that will be used to translate the body of a method `M` in `C`. They associate (i) bound descriptors to the static fields of the class, (ii) descriptors using only type variables of the class to friend types/methods of the instance field `td`, and (iii) descriptors using type variables of the method to friend types/methods of the method descriptor $\mathtt{e}_m$ of the current VPMT. Then, the type of `this` is associated to the the local type descriptor `td`, and the signature of the current descriptor is associated to the formal argument `md`.

The translation for a method follows the pattern of FGJ. The only difference here, is that we have to add an extra argument that will contain the position of the method descriptor in the VPMT. The environments $\Theta^{\mathtt{C},\mathtt{M}}$ and $\Pi^{\mathtt{C},\mathtt{M}}$ created in this way will then be used to properly translate the expressions contained in the method `M` of class `C`.

4.8 Translating expressions

The rules defining translation of expressions are shown in Figure 4. The translation for a variable `x` and for a field accessing `e.f` is the same as in FGJ. A cast `(T)e` is translated so as to invoke the method `cast` on the descriptor for `T`, passing the translation of `e`. The allocation expression `new N(`$\overline{\mathtt{e}}$`)` is translated by passing as first argument the descriptor for `N`. Analogously, in method invocation we pass as first argument the position of the method descriptor in its VPMT.

5. AN EXAMPLE

In order to allow a better understanding of our formalization, here we provide a comprehensive example of translation. Our source code is shown in Figure 5. It contains a FGJ class `Pair` with two type parameters, two corresponding fields, and five methods doing several things. The corresponding translation according to our formalization is provided in Figure 6; The class `Pair` has the bound type `Pair<Object,Object>` and no bound methods, so the translated class will just have the static field `fST_0`. The translation of the constructor directly follows from the definition. The class `Pair<R,S>` has the free type `Pair<S,R>` (used by the method `reverse`) and the free method signature `Pair<R,S>.reverse<>()` (used in the method `chgSecond`), from which follows the method `createTD` in the translated code.

Then, each method has its own free types and free method signatures. In particular, `Pair<R,S>.chgFirst<T>` has the friend type `Pair<T,S>`, while `Pair<R,S>.chgSecond<T>` has the friend type `Pair<T,R>` (receiver of the latter invocation of `reverse`), and the friend methods `Pair<S,R>.chgFirst<T>` and `Pair<T,R>.reverse()`. Correspondingly, in the translated class we have the method `createMD`

[2]Such a method supports the inspection of the descriptor of an object.

```
class Pair<R extends Object,S extends Object> extends Object{
    R r;
    S s;
    Pair(R r,S s){ super();this.r=r;this.s=s;}
    <> Pair<S,R> reverse(){ return new Pair<S,R>(this.s,this.r);}
    <> Pair<Object,Object> getOO(){return new Pair<Object,Object>(new Object(),new Object());}
    <T> Pair<T,S> chgFirst(T t){ return new Pair<T,S>(t,this.s);}
    <T> Pair<R,T> chgSecond(T t){ return this.reverse<>().chgFirst<T>(t).reverse<>();}
    <> R castToFirst(Object o){ return (R)o;}
}
```

Figure 5: Example of source code

```
class Pair extends LMObj{

    static TD fST_0=Pair.createTD(new $TD[]{LMObj.createTD[]{},LMObj.createTD[]{}});

    Object r; Object s;

    $TD td; $TD getTD(){ return td;}
    Pair($TD td,Object r,Object s){ super(td.father);this.r=r;this.s=s;}

    static $TD createTD($TD[] p){
      $TD t=$TDM.register(Pair.class,p);
      if (t.checkNew()){
        t.father=LMObj.createTD(new $TD[]{});
        t.friendTs=new $TD[]{  Pair.createTD(new $TD[]{p[1],p[0]})  };
        t.friendMs=new int[]{  Pair.createMD(new $TD[]{p[0],p[1]},0,new $TD[]{})};
        t.initVPMT();
      }
      return t;
    }

    static int createMD($TD t,int pos,$TD[] p){
      $MD m=$MDM.register(t,pos,p);
      if (m.checkNew()){
        t.VPMT[pos].addElement(m);
        m.lPos=t.VPMT[pos].size()-1;
        if (pos==0) {   t.friendTs=new $TD[]{};
                        t.friendMs=new int[]{};  }
        if (pos==1) {   t.friendTs=new $TD[]{};
                        t.friendMs=new int[]{};  }
        if (pos==2) {   t.friendTs=new $TD[]{ Pair.createTD(new $TD[]{p[0],td.friendTs[1]})};
                        t.friendMs=new int[]{};  }
        if (pos==3) {   t.friendTs=new $TD[]{ Pair.createTD(new $TD[]{p[0],td.param[0]})};
                        t.friendMs=new int[]{ Pair.createMD(td.friendTs[0],2,new $TD[]{p[0]}),
                                              Pair.createMD(t.friendTs[0],0,new $TD[]{})     }; }
        if (pos==4) {   t.friendTs=new $TD[]{};
                        t.friendMs=new int[]{};  }
        $MDM.propagate(m);
      }
      return m;
    }

    Pair reverse(int md){ return new Pair(td.friendTs[0],this.s,this.r);}

    Pair getOO(int md){return new Pair(fST_0,new Object(),new Object());}

    Pair chgFirst(int md,Object t){
        return new Pair((($MD)(td.VPMT[2].elementAt(md))).friendTs[0],t,this.s);}

    Pair chgSecond(int md,Object t){
        return this.reverse(td.friendMs[0])
                   .chgFirst((($MD)(td.VPMT[2].elementAt(md))).friendMs[0],t)
                   .reverse((($MD)(td.VPMT[2].elementAt(md))).friendMs[1]);}

    Object castToFirst(Object o){ return td.p[0].cast(o);}
}
```

Figure 6: Translation of the source code

as shown in Figure 6.

Finally, the translation of methods simply proceeds as follows. In the signature, argument types and return type are erased, and one argument of type `int` is added. Then, the translation of the returning expression extends the one done for FGJ, but we pass the type descriptor in the `new` expressions, the position of method descriptor in method calls, and we exploit the method `$TD.cast` for implementing casts. Bound type descriptors and method descriptors are accessed through the static fields of the class, free types and methods of the class through the friends of the local type descriptor `td`, and free types and methods of the methods through friends of the current method descriptor. The latter is obtained accessing the VPMT of `td` and exploiting the formal parameter `md`, by the expression `(($MD)(td.VPMT[l].elementAt(md)))` where `l` is the (static) position of the method in the VPMT, say 0 for `reverse`, 1 for `getOO`, 2 for `chgFirst`, and so on.

6. FROM FORMALIZATION TO IMPLEMENTATION

We think it is interesting to discuss how we are going to develop an implementation of LM out from the formalization here introduced. First of all, we rely on a free tool that builds parsers and/or tree generators from source specification files containing BNF-like grammars, which is Sun's JavaCC product [7]. The main features of this tool is that it produces Java code, and permits to insert pieces of Java code in the source specification, allowing to have a fine-grained control on the parsing process and on the shape of the generated tree. Then, the JavaCC distribution also includes the source for creating a parser for Java 1.2.

Our goal here is not to produce an actual implementation suitable for a large-scale release. Instead, the objective is to obtain a prototype of LM translator, which can be used as a tool supporting the measurements of the performance of the translated code. In fact, since relevant performance issues concerns memory footprint and load-time overhead, we need to translate medium-large benchmarks, so we can't just rely on small applications translated by-hand.

Also, we would like to obtain an implementation allowing for the fast prototyping of new versions of the translation. In fact, we believe that the process of measuring performance will give feedbacks motivating an appropriate tuning of the translation. Basically, we intend to address this issue by clearly encapsulating the part of the translator dealing with the key concepts related to parametric polymorphism, i.e., the part implementing the formalization provided in this paper.

As our target is a prototype, we don't need it to be fully-featured. The language accepted is broadly similar to the one accepted by GJ. The two basic differences are (i) that we won't allow to omit type parameters specification from method invocation[3], and (ii) we won't deal with raw types (see [6] for details on these issues). Producing a fast translation process is not a primary goal. Then, we don't need our translator to intercept all the kinds of error due to an incorrect program. We accept that errors can be caught by either the translator or by the successive actual compilation of the resulting Java file. As a result, we won't issue any constraint on how the errors intercepted by the translator are notified to the user.

[3]Type parameter inference is not possible, in general, in the implementations supporting parametric types at run-time

The need for carefully studying a methodology for building the translator comes in from the fact that the complete syntax of Java is huge, so the translator will be a very complex system. Two arguments suffice in emphasizing this: the tree generator for Java, as created by JavaCC, is about 9000 lines of Java code, and the number of different nodes of the tree, which is basically the number of cases we have to consider during the translation, is bigger than 50.

It worth noting that the methodology we will discuss here is not strongly tested, and it should basically considered a proposal for addressing the specification here discussed. When the implementation will be finished we expect to have gathered feedbacks and experience for improving the methodology and describing its details. The important thing here, is our claim that the way we formalized the translation in this paper is the most suitable approach for supporting the actual implementation of the prototype. Basically, it plays a crucial role in encapsulating the core of the translation, allowing changes to be limited to this part, and allowing to build the actual translator on top of it in an incremental way.

1. The first step of the process is to obtain a trivial translator for the source language, translating a source code written in Java with parametric polymorphism in itself. Thanks to the JavaCC tool and the JavaCC specification file of Java 1.2 provided with the distribution, this can be simply done as follows:

   - Creating a suitable tree generator from the JavaCC specification file of Java 1.2, by *decorating* it with the appropriate insertion of Java code.
   - Adding the syntax needed to support parametric polymorphism.
   - Defining a simple visitor for the tree, re-producing the source code given as input.

   The basic goal of this step is to immediately produce the JavaCC specification file of the full source language, which helps in understanding the complexity of the domain and in enumerating all the language constructs we will have to deal with. Later, this will help in deciding what is the better order for incrementally building the translator on the top of its core.

2. The second step is to produce the formalization of the translation. In particular it is important that doing this we focus on all the important issues of the translation, and abstract away from details that are conceptually unnecessary. The one given in the paper fulfills both the requirements, but left out, mostly for space reasons, important aspects such as parametric interfaces and inner classes.

3. The third step is to implement a one-to-one translator taking a source file containing a closed set of FGJ classes and producing a valid Java file. This turns out to be a mere programming exercise, as the source language is very simple and its translation behaviour has been precisely described in the formalization. What we obtain from this step, is the core of the translator.

   At the current time, our development process reached this point, that is, we already have a translator for FGJ to Java, which for instance, can be used to translate our example of Section 5. So, the next steps actually describe what we meant to do for obtaining the full prototype.

4. The following step, is to adapt this translation so as to produce a somewhat fully-featured translator for FGJ into Java, comprehending the management of multiple source files, and dealing with packages and package-qualified class names. The only difference between the result of this step and the final prototype is on the "size" of the source language.

5. The translator produced so far has to be carried from accepting FGJ files to full Java files providing parametric polymorphism, by adding one language constructs at a time. Doing this we exploit the specification file obtained in Step 1, basically adding one BNF rule each time. The most appropriate way of doing this is, orderly:

   - adding the management of class members left-out by FGJ, such as inner classes and constructors, as well as the management of interfaces and static members;
   - adding primitive types and arrays;
   - adding the remaining constructs about expressions;
   - adding the remaining constructs about statements;
   - adding the management of other issues, such as field shadowing and method overriding.

   Where possible, each addition should be tested and debugged alone.

6. Finally, remaining orthogonal issues can be addressed, such as dealing with the inspection of external classes that are used by the code we are translating, but whose source code is not available. These can be either legacy Java classes or LM-generated classes.

7. CONCLUSIONS

The contribution of this paper is twofold. On the one hand, it provides for a compact yet comprehensive formalization of LM translator, a proposal for extending Java with parametric polymorphism by means of a translation. As the LM translator is a complex system, whose informal description tends to be long and typically requires to provide many low level details, the formalization here introduced is an unavoidable tool for its complete and precise understanding. On the other hand, we described how such a formalization can help in quickly building a core prototype which later can be incrementally extended so as to lead to the final implementation.

Our current and future work is devoted on applying our methodology for implementing LM translator, and to go further with its formalization. In fact, the one given in this paper cannot be applied for directly proving properties such as type preservation. By reducing the scope of the target language we should be able to follow the same approach taken in [3]. In particular, it should be possible to define a translation covering a sufficiently large subset of LM behaviour by means of a compilation of FGJ into an extension of FJ with side-effects (field assignment) and a minimal support of reflection.

8. REFERENCES

[1] G. Bracha. *Adding Generic Types to the JavaTM Programming Language*. Java Specification Request, JSR-000014, http://java.sun.com, 1998.

[2] C. Cartwright and G. Steele. Compatible genericity with run-time types for the Java programming language. In *Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 201–215. ACM, October 1998.

[3] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java, a minimal core calculus for Java and GJ. In *Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 132–146. ACM, October 1999.

[4] M.Ellis and B.Stroustroup. *The Annotated C++*. Addison-Wesley, 1990.

[5] M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In *Symposium on Principles of Programming Languages*, pages 146–159. ACM, 1997.

[6] M. Odersky, P. Wadler, G. Bracha, and D. Stoutamire. Making the future safe for the past: Adding Genericity to the Java programming language. In *Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 183–200. ACM, October 1998.

[7] Sun Microsystem. JavaCC 2.0. Distributed by Metamata, http://www.webgain.com/products/metamata/java_doc.html.

[8] Sun Microsystems. *JSR-14 Public Draft*. http://java.sun.com, 2001.

[9] D. Syme and A. Kennedy. Design and implementation of generics for the .NET Common Language Runtime. In *proceedings of Programmin Languages Design and Implementation (PLDI2001)*. ACM, June 2001.

[10] M. Viroli. Parametric polymorphism in Java: an efficient implementation for parametric methods. In *Symposium on Applied Computing (SAC)*, pages 610–619. ACM, March 2001.

[11] M. Viroli and A. Natali. Parametric Polymorphism in Java: an approach to translation based on reflective features. In *Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 146–165. ACM, Oct 2000.