A Framework for Object-Oriented Program Analyses via *Core-Java*

Florin Craciun Hong Yaw Goh Wei-Ngan Chin
Department of Computer Science, National University of Singapore
{craciunm|gohhy|chinwn}@comp.nus.edu.sg

Abstract

A common practice for a rapid prototyping of an object-oriented program analysis is to define a lightweight fragment of Java, that is sufficiently small to facilitate a rigorous analysis of key properties. Due to a lack of important Java features, an experimental evaluation on real-world code is not easy. The solution is either to extend the prototype to the whole Java or to rewrite the real-world code in the lightweight language. This paper proposes an intermediate solution through Core-Java*, a core calculus for Java and a comprehensive set of translation rules from Java to* Core-Java*. The translation can be guided by the specific requirements of each program analysis. We have built an implementation of our framework and have used it for two different analyses on Java programs.*

1. Introduction

We propose a framework for rapidly prototyping of various type-based analyses for object oriented languages like Java. Java has a lot of features, a large syntax with a complicated semantics. Our framework consists of a minimal core calculus for Java, called *Core-Java* and set of rules that permit a type-based source-to-source transformation of Java programs into Core-Java programs.

Core-Java is designed in the same minimalist spirit as Featherweight Java [10], Middleweight Java [3] or Concurrent Java [7]. In contrast to the main motivations for these proposals, we are interested in a more practical core calculus that incorporates all Java features [8] and makes it easy to analyze and manipulate real-world Java programs.

We are targeting applications that intend to carry out analyses and source-to-source transformations on Java programs. A good intermediate language for such a task should be simple to analyse, close to source and able to handle real-world code [11].

However, our *Core-Java* language is still more abstract than typical intermediate languages designed for compilation. Many of the intermediate languages lose structural information about types, loops and other high-level constructs.

In essence, *Core-Java* is an expression-oriented language, that makes easier the formulation of static and dynamic semantics. Expression languages are more suitable for type-based analyses that work on an abstract syntax tree rather than on a control flow graph. However, the language allows constructions that control the program flow.

We designed a comprehensive set of translation rules from Java to *Core-Java*. The output of the translator consists of the *Core-Java* program and some other data structures like the global dependency graph that can further help the program analyses. The translation can be guided by the specific requirements of each program analysis.

The paper is structured as follows. Section 2 describes the syntax of *Core-Java*, Section 3 presents the set of the translation rules from Java to *Core-Java* and Section 4 discusses the support for program analyses. Finally, we present experimental results in Section 5 and conclude in Section 6.

2. *Core-Java* Language

$$
\begin{array}{rcl}
P & ::= & \texttt{package}\ n\ (\texttt{import}\ n)^*\ \mathit{def}^* \\
\mathit{def} & ::= & \mathit{mdf}^*\ \texttt{class}\ c_1\ \texttt{extends}\ c_2\ \texttt{implements}\ c^*\ \{\mathit{fld}^*\ \mathit{meth}^*\} \\
 & & \mid\ \mathit{mdf}^*\ \texttt{interface}\ c_1\ \texttt{extends}\ c^*\ \{\mathit{meth}^*\} \\
\tau & ::= & c \mid \mathit{prim} \mid \mathit{void} \\
\mathit{fld} & ::= & \mathit{mdf}^*\ \tau\ f \\
\mathit{meth} & ::= & \mathit{mdf}^*\ \tau\ \mathit{mn}\,((\tau\ v)^*)\ \texttt{throws}\ c^*\ \mathit{eb} \\
x & ::= & v \mid k \\
w & ::= & v \mid v.f \mid c.f \\
\mathit{eb} & ::= & \{(\tau\ v)^*\ e\} \\
e & ::= & k \mid w \mid (c)null \mid \mathit{eb} \mid () \mid w = e \mid e_1\,;e_2 \mid (\tau)v \\
 & & \mid v.\mathit{mn}(x^*) \mid c.\mathit{mn}(x^*) \mid \texttt{new}\ c(x^*) \mid \texttt{if}\ x\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \\
 & & \mid \texttt{return}\ x \mid \texttt{continue} \mid \texttt{break} \mid \texttt{while}\ x\ \mathit{eb} \\
 & & \mid \texttt{throw}\ v \mid \texttt{try}\ e\ \texttt{catch}\ (c\ v\ \mathit{eb})^* \mid \texttt{synchronized}\ v\ \texttt{in}\ e
\end{array}
$$

$v \in$ **variable names** $c \in$ **class or interface names**
$f \in$ **field names** $\mathit{mn} \in$ **method names**
$k \in$ **constants** $\mathit{prim} \in$ **primitive types**
$n \in$ **package names** $\mathit{mdf} \in$ **modifiers (e.g. static, public, etc.)**

Figure 1. The Syntax of *Core-Java*

Figure 1 presents the syntax of our *Core-Java* language.

Our language features a reduced number of syntactic and conceptual forms as compared to the Java language. We designed *Core-Java* as an *expression-oriented* language with *object-oriented features*. This section describes the main language and then next sections discuss how to minimise features that are required by different kinds of program analyses. A *Core-Java* program, P consists of a set of class and interface declarations, def. To support separate compilation and to help the name look-up mechanism, a program also contains the name of the package in which it is defined and the list of the imported classes. Class names follow the same convention as in Java, either simple, consisting of a single identifier, or qualified, consisting of a sequence of identifiers separated by "." tokens. The language supports simple inheritance by extending a class, but also multiple inheritance through the interface mechanism. An interface is an abstract class that only contains methods without body. A class contains the declared fields and methods and the inherited fields and methods with method overriding directly supported. Modifiers are supported at class, field and method level. They are mainly used to control access or to declare the static class components. A method declaration $meth$ consists of the method name, its return type, its raised exceptions, number and types of its arguments, method modifiers, and a block expression for the method body. *Core-Java* uses by default pass-by-value mechanism, but it also supports pass-by-reference mechanism used in languages like C#. There is a distinction between a block expression, eb and an expression, e. A block expression $\{(\tau\ v)^*e\}$ consists of a list of local variable declarations and a body e, that is an expression. A block expression is evaluated to its body. The notation w denotes a lvalue, a value that can appear on the left-hand side of an assignment. A lvalue can be either a variable, v, or an object field, $v.f$ or a static field, $c.f$. Note that field access is restricted to the simplest form, $v.f$. Java keywords *this* and *super* are treated as special variable names in *Core-Java*. The notation x denotes either a variable or a constant value and is used to restrict some expressions (e.g. method arguments, returned expression, etc). *Core-Java* does not contain statements as it contains only expressions. Expressions, e include assignments, cast operations, expression blocks, method invocations, conditionals, loops, concurrency primitives, exceptions and control-flow primitives. The notation () denotes an empty expression and its type is *void*. An assignment evaluates to a *void* value, a sequence $e_1;e_2$ evaluates to the value of expression e_2, and a loop also evaluates to *void*. Static fields and methods are treated similar with the non-static class components. However, when the receiver object is not available, its place is replaced by the class name, $c.f$ and $c.mn(x^*)$. A `new` $c(x^*)$ invokes a default constructor of the c class. A constructor is a non-static method having the same name as its class. *Core-Java* monomorphic type, τ can be either a class type, or *void* or any Java primitive type.

3. Java to *Core-Java* Translation

The goal of our translation is to automatically generate a *Core-Java* program from a given Java program. We assume that Java input program is *well-typed*. This section describes the main rules of Java to *Core-Java* translation. We formulate the rules as a type-directed translation that follows the syntax of the Java source language. The rules are *type-preserving*, that is, they guarantee that both programs, the Java input and the *Core-Java* output, have the same Java type. Our algorithm consists of three main steps.

Generating Descriptor of Compilation Unit

A compilation unit can be any Java individual file that contains class or interface declarations. For each compilation unit, we generate its attached descriptor file. The syntax of a descriptor file is shown in Figure 2.

$$
\begin{array}{rcl}
P & ::= & \texttt{package}\ n\ (\texttt{import}\ n)^*\ \mathit{def}^* \\
\mathit{def} & ::= & \mathit{mdf}^*\ \texttt{class}\ c_1\ \texttt{extends}\ c_2\ \texttt{implements}\ c^*\ \left\{(\mathit{mdf}^*\ \tau\ f)^*\ \mathit{meth}^*\right\} \\
& | & \mathit{mdf}^*\ \texttt{interface}\ c_1\ \texttt{extends}\ c^*\ \left\{\mathit{meth}^*\right\} \\
\mathit{meth} & ::= & \mathit{mdf}^*\ \tau\ \mathit{mn}\ ((\tau\ v)^*)\ \texttt{throws}\ c^*
\end{array}
$$

Figure 2. Descriptor Syntax

A descriptor consists of typing information of each interface, class and their fields and methods, but without the method body. The descriptor also contains the class type dependencies, that include the direct superclass, the implemented interfaces, the types of the fields, the headers of the methods, the types from the bodies of the methods and so forth. All these dependencies are added to the initial `import` list of the Java input file. Type class dependencies correspond to the standard type environment from [2], but can be refined using the techniques of separate compilation [1].

Global Dependency Graph

Our translation is required to process the class and interface declarations in some particular order given by the complex inter-dependency among classes, interfaces and methods. The dependency graph has the class and interface declarations organised into a hierarchy of strongly connected components (SCCs). Each set of declarations in a SCC is regarded as a set of mutually-dependent declarations, that are processed at the same time. Through a bottom-up processing of each SCC, we perform the translation in a systematic fashion. The global dependency graph is also kept after the translation to be used by the subsequent program analyses. Furthermore, for program analyses that are done at a per method level, we require finer-grained dependencies like method to class, method to interface, method to method and even class or interface to method due to method overriding.

Translation

The type-based translation is formulated as a type inference for the Java input program. The type system behind translation is similar with that formalised in [6]. For sake of brevity, we simplified the presentation of the translation rules. Figure 3 and Figure 4 present a fragment of our translation rules. We note by *D* the set of descriptors required to type check a Java program and by Γ the current type environment. The rules use a monadic function *fresh* to generate unique variable names. The main judgments employed by our translation rules are presented below.

$$\frac{D \vdash \mathit{def}_i \ \Rightarrow_{\mathit{def}} \mathit{def}'_i \ \ i = \overline{1,n}}{D \vdash \texttt{package}\ n (\texttt{import}\ n)^* \mathit{def}_{i=\overline{1,n}} \Rightarrow_{\mathit{prg}} \texttt{package}\ n (\texttt{import}\ n)^* \mathit{def'}_{i=\overline{1,n}}}\ [\mathbf{Unit}]$$

$$\frac{\begin{array}{c} D \vdash \mathit{fld}_i \Rightarrow_{\mathit{fld}} \mathit{fld'}_i, eb_i \quad \mathit{addToDefaultConstructor}(eb_i)\ i = \overline{0,n} \\ D, \{\mathit{this} : \mathit{cn}, \mathit{super} : \mathit{cn}'\} \vdash \mathit{meth}_i \Rightarrow_{\mathit{meth}} \mathit{meth}'_i \quad i = \overline{0,p} \end{array}}{\begin{array}{c} D \vdash \mathit{mdf}^* \texttt{class}\ \mathit{cn}\ \texttt{extends}\ \mathit{cn}' \texttt{implements}\ c_{i=\overline{0,k}} \{\mathit{fld}_{i=\overline{0,n}} \mathit{meth}_{j=\overline{0,p}}\} \\ \Rightarrow_{\mathit{def}} \mathit{mdf}^* \texttt{class}\ \mathit{cn}\ \texttt{extends}\ \mathit{cn}' \texttt{implements}\ c_{i=\overline{0,k}} \{\mathit{fld'}_{i=\overline{0,n}} \mathit{meth'}_{j=\overline{0,p}}\} \end{array}}\ [\mathbf{Class{-}Decl}]$$

$$\frac{D, \{\} \vdash e \Rightarrow_{\mathit{exp}} e' : \tau}{\begin{array}{c} D \vdash \mathit{mdf}^*\ \tau\ f = e \\ \Rightarrow_{\mathit{fld}} \mathit{mdf}^*\ \tau\ f, \{e'\} \end{array}}\ [\mathbf{Fld{-}Decl}] \qquad \frac{\Gamma' = \Gamma, (v_j : \tau_j)_{j=\overline{0,p}} \quad D, \Gamma' \vdash eb \Rightarrow_{\mathit{stm}} eb'}{\begin{array}{c} D, \Gamma \vdash \mathit{mdf}^*\ \tau\ \mathit{mn}((\tau_j\ v_j)_{j=\overline{0,p}}) \texttt{throws}\ c^*\ eb \\ \Rightarrow_{\mathit{meth}} \mathit{mdf}^*\ \tau\ \mathit{mn}((\tau_j\ v_j)_{j=\overline{0,p}}) \texttt{throws}\ c^*\ eb' \end{array}}\ [\mathbf{Meth{-}Decl}]$$

$$\frac{\begin{array}{c} \Gamma' = \Gamma, (v_j : \tau_j)_{j=\overline{0,p}} \quad D, \Gamma' \vdash eb \Rightarrow_{\mathit{stm}} eb' \\ eb'' = \{\texttt{synchronized}\ \mathit{this}\ \texttt{in}\ eb'\} \end{array}}{\begin{array}{c} D, \Gamma \vdash \texttt{synchronized} \tau\ \mathit{mn}((\tau_j\ v_j)_{j=\overline{0,p}}) \texttt{throws}\ c^*\ eb \\ \Rightarrow_{\mathit{meth}}\ \tau\ \mathit{mn}((\tau_j\ v_j)_{j=\overline{0,p}}) \texttt{throws}\ c^*\ eb'' \end{array}}\ [\mathbf{Sync{-}Meth{-}Decl}]$$

$$\frac{\begin{array}{c} v',\ v'' = \mathit{fresh}() \\ (\mathit{this}{:}\mathit{cn}) \in \Gamma \quad \Gamma' = \Gamma, (v_j : \tau_j)_{j=\overline{0,p}} \quad D, \Gamma' \vdash eb \Rightarrow_{\mathit{stm}} eb' \\ eb'' = \{\mathit{Class}\ v;\ v = \mathit{Class.forName(cn)};\ \texttt{synchronized}\ v'\ \texttt{in}\ eb'\} \\ eb''' = \{\texttt{try}\ eb''\ \texttt{catch}(\mathit{ClassNotFoundException} v'' \{\})\} \end{array}}{\begin{array}{c} D, \Gamma \vdash \texttt{static}\ \texttt{synchronized} \tau\ \mathit{mn}((\tau_j\ v_j)_{j=\overline{0,p}}) \texttt{throws}\ c^*\ eb \\ \Rightarrow_{\mathit{meth}}\ \texttt{static}\ \tau\ \mathit{mn}((\tau_j\ v_j)_{j=\overline{0,p}}) \texttt{throws}\ c^*\ eb''' \end{array}}\ [\mathbf{Sync{-}Static{-}Meth{-}Decl}]$$

Figure 3. Fragment of our Translation Rules

Compilation Unit

$D \vdash U \Rightarrow_{\mathit{prg}} P$ denoting the translation of a Java compilation unit *U* into a *Core-Java* program *P*, where *U* and *P* are well-defined with respect to the set of descriptors, *D*. This judgment is defined as rule [Unit] in Figure 3.

Class and Interface Declaration

$D \vdash \mathit{def} \Rightarrow_{\mathit{def}} \mathit{def'}$ denoting the translation of a Java class/interface declaration *def* into a *Core-Java* class/interface declaration def', where *def* and *def'* are well-defined with respect to the set of descriptors, *D*. This judgment is defined as rule [Class−Decl] in Figure 3, that invokes the translation rules for fields and methods. A field may have an initializer, which is translated into a block. All these block expressions are collected into a new *Core-Java* class method that is implicitly invoked by each class constructor. The type environment of the method rules is initialized with the types of the special variables *this* and *super*. In case of a Java interface declaration, the possible static fields are moved into the *Core-Java* declaration of each class that implements that interface. For simplicity, this operation is omitted from the above formalism.

Field Declaration

$D \vdash \mathit{fld} \Rightarrow_{\mathit{fld}} \mathit{fld'}, \mathit{eb'}$ denoting the translation of a Java field declaration fld into a *Core-Java* field declaration *fld'*, where *fld* and *fld'* are well-defined with respect to the set of descriptors, *D*. A possible initializer of a Java class field is translated into a *Core-Java* block expression, *eb'*. The judgment is defined as rule [Fld−Decl] in Figure 3.

Method Declaration

$D, \Gamma \vdash \mathit{meth} \Rightarrow_{\mathit{meth}} \mathit{meth'}$ denoting the translation of a Java method declaration *meth* into a *Core-Java* method declaration *meth'*, where *meth* and *meth'* are well-defined with respect to the set of descriptors, *D* and type environment Γ. The class constructors are treated in a similar manner as the methods. The overloaded Java methods are consistently renamed by translation such that in *Core-Java* there are no overloaded methods. For simplicity, renaming is omitted from the above formalism. The judgment is used by the rules [Meth−Decl], [Sync−Meth−Decl] and [Sync−Static−Meth−Decl] in Figure 3. These rules invoke the translation rule for the method body with a type environment completed with the method arguments. The last two rules translate the `synchronized` methods. A synchronized method acquires a monitor before it executes. For static methods, the monitor associated with the method's $Class$ object is used. For an instance method, the monitor associated with *this* is used.

Statement

$D, \Gamma \vdash \mathit{stm} \Rightarrow_{\mathit{stm}} e$ denoting the translation of a Java statement *stm* into a *Core-Java* expression e, where *stm* and e are well-defined with respect to the type environment, Γ and the set of descriptors, *D*. The rules [Local−Var−Decl], [IF] and [While] in Figure 4 use this judgment. Rule [Local−Var−Decl] moves the initializers of the local variables after the variable declarations, while rule [IF] replaces the predicate of conditional with a fresh `boolean` variable. Rule [While] simplifies the condition to a variable and then translates the loop body. *Core-Java* supports directly the `while` loop, since all other kinds of Java loops are converted to `while`. Note that the rules in Figure 4 use the first two rules, [Cnv−1] and [Cnv−2] to introduce a temporary variable only if needed.

Expression

$D, \Gamma \vdash e \Rightarrow_{\mathit{exp}} e' : \tau$ denoting the translation of a Java expression e into a *Core-Java* expression e', where e and e' have the type τ with respect to the type environment, Γ and the set of descriptors, *D*. The last eight rules in Figure 4 are based on this judgment. Rule [Meth−Invoke] replaces all method parameters (including the receiver), which are not variables or constants, with fresh variables. *Core-Java* does not have operators on primitive types, instead it contains a

$$\left[\underline{\mathbf{Cnv-1}}\right] \quad \frac{}{\vdash(\tau,x)\Rightarrow_{\mathit{cnv}}((),(),x)} \qquad \left[\underline{\mathbf{Cnv-2}}\right] \quad \frac{v=\mathit{fresh}()}{\vdash(\tau,e)\Rightarrow_{\mathit{cnv}}(\tau\ v, v{=}e, v)}$$

$$\left[\underline{\mathbf{Local-Var-Decl}}\right]$$
$$\frac{D,\Gamma\vdash e\Rightarrow_{\mathit{exp}}e' : \tau \quad \Gamma'{=}\Gamma,(v:\tau) \quad D,\Gamma'\vdash \mathit{stm}_i\Rightarrow_{\mathit{stm}}e'_i \ i=\overline{0,n}}{D,\Gamma\vdash\{(\tau\ v{=}e);\mathit{stm}_{i=\overline{0,n}}\}\Rightarrow_{\mathit{stm}}\{\tau\ v; v{=}e'; e'_{i=\overline{0,n}}\}}$$

$$\left[\underline{\mathbf{IF}}\right]$$
$$\frac{\begin{array}{c}D,\Gamma\vdash e\Rightarrow_{\mathit{exp}}e':\texttt{boolean} \quad \vdash(\texttt{boolean},e')\Rightarrow_{\mathit{cnv}}(d',e'',v)\\ D,\Gamma\vdash \mathit{stm}_i\Rightarrow_{\mathit{stm}}e'_i \ \ i=\overline{1,2}\end{array}}{D,\Gamma\vdash\texttt{if}\ e\ \mathit{stm}_1\ \texttt{else}\ \mathit{stm}_2\Rightarrow_{\mathit{stm}}\{d';e'';\texttt{if}\ v\ \texttt{then}\ e'_1\ \texttt{else}\ e'_2\}}$$

$$\left[\underline{\mathbf{While}}\right]$$
$$\frac{\begin{array}{c}D,\Gamma\vdash e\Rightarrow_{\mathit{exp}}e':\texttt{boolean} \quad \vdash(\texttt{boolean},e')\Rightarrow_{\mathit{cnv}}(d',e'',v)\\ D,\Gamma\vdash \mathit{stm}\Rightarrow_{\mathit{stm}}eb\end{array}}{D,\Gamma\vdash\texttt{while}\ e\ \mathit{stm}\Rightarrow_{\mathit{stm}}\{d';e'';\texttt{while}\ v\ \{eb;\ e''\}\}}$$

$$\left[\underline{\mathbf{Meth-Invoke}}\right]$$
$$\frac{\begin{array}{c}D,\Gamma\vdash e_0\Rightarrow_{\mathit{exp}}e'_i:\tau_i \quad \vdash(\tau_i,e'_i)\Rightarrow_{\mathit{cnv}}(d'_i,e''_i,v_i) \quad i=\overline{0,n}\\ D\vdash(\tau\ m(\tau_1..\tau_n)\ldots)\in\tau_0\end{array}}{D,\Gamma\vdash e_0.m(e_1..e_n)\Rightarrow_{\mathit{exp}}\{d_{i=\overline{0,n}};e''_{i=\overline{0,n}};v_0.m(v_1..v_n)\}:\tau}$$

$$\left[\underline{\mathbf{Binary-Op}}\right]$$
$$\frac{\begin{array}{c}D,\Gamma\vdash e_i\Rightarrow_{\mathit{exp}}e'_i:\tau_i \quad \vdash(\tau_i,e'_i)\Rightarrow_{\mathit{cnv}}(d'_i,e''_i,v_i) \quad i=\overline{1,2}\\ \texttt{op}{=}\mathit{getPrimitive}(\mathit{BinaryOp},\tau_1,\tau_2) \quad D\vdash(\tau\ \texttt{op}(\tau_1,\tau_n)\ldots)\in\texttt{Primitives}\end{array}}{D,\Gamma\vdash e_1\ \mathit{BinaryOp}\ e_2\Rightarrow_{\mathit{exp}}\{d_1;d_2;e''_1;e''_2;\texttt{Primitives.op}(v_1,v_2)\}:\tau}$$

$$\left[\underline{\mathbf{As-1}}\right] \quad \frac{\begin{array}{c}D,\Gamma\vdash e_1\Rightarrow_{\mathit{exp}}\{d;e;w\}:\tau\\ D,\Gamma\vdash e_2\Rightarrow_{\mathit{exp}}e'_2:\tau_2\end{array}}{D,\Gamma\vdash e_1{=}e_2\Rightarrow_{\mathit{exp}}\{d;e;w{=}e'_2\}:void} \qquad \left[\underline{\mathbf{As-2}}\right] \quad \frac{\begin{array}{c}D,\Gamma\vdash e_1\Rightarrow_{\mathit{exp}}w:\tau\\ D,\Gamma\vdash e_2\Rightarrow_{\mathit{exp}}e'_2:\tau_2\end{array}}{D,\Gamma\vdash e_1{=}e_2\Rightarrow_{\mathit{exp}}w{=}e'_2:void}$$

$$\left[\underline{\mathbf{LVal-1}}\right] \quad \frac{\begin{array}{c}D,\Gamma\vdash e\Rightarrow_{\mathit{exp}}e':\tau\\ D\vdash(\tau_1 f_1)\in\tau \quad D\vdash(\tau_2 f_2)\in\tau_1\end{array}}{\begin{array}{c}D,\Gamma\vdash e.f_1.f_2\Rightarrow_{\mathit{exp}}\\ \{\tau\ v;\tau_1\ v_1;v{=}e';v_1{=}v.f_1;v_1.f_2\}:\tau_2\end{array}} \qquad \left[\underline{\mathbf{LVal-2}}\right] \quad \frac{(w:\tau)\in\Gamma}{D,\Gamma\vdash w\Rightarrow_{\mathit{exp}}w:\tau}$$

$$\left[\underline{\mathbf{BOp-As-1}}\right] \quad \frac{\begin{array}{c}D,\Gamma\vdash e_1\Rightarrow_{\mathit{exp}}\{d;e;w\}:\tau\\ D,\Gamma\vdash e_1\ \mathit{BinaryOp}\ e_2\Rightarrow_{\mathit{exp}}e'_2:\tau_2\end{array}}{\begin{array}{c}D,\Gamma\vdash e_1\ \mathit{BinaryOp}{=}e_2\\ \Rightarrow_{\mathit{exp}}\{d;e;w{=}e'_2\}:void\end{array}} \qquad \left[\underline{\mathbf{BOp-As-2}}\right] \quad \frac{\begin{array}{c}D,\Gamma\vdash e_1\Rightarrow_{\mathit{exp}}w:\tau\\ D,\Gamma\vdash e_1\ \mathit{BinaryOp}\ e_2\Rightarrow_{\mathit{exp}}e'_2:\tau_2\end{array}}{\begin{array}{c}D,\Gamma\vdash e_1\ \mathit{BinaryOp}{=}e_2\\ \Rightarrow_{\mathit{exp}}w{=}e'_2:void\end{array}}$$

Figure 4. Fragment of our Translation Rules

special primitive class, called *Primitives*. This class contains a corresponding static method for each primitive operator to avoid the operator overloading. Rule [Binary−Op] illustrates the use of class *Primitives* for binary operators. Rules [As−1] and [As−2] translate an assignment, first its left hand side and then its right hand side. Note that, the value of a *Core-Java* block expression is the value of the last expression of the block body. Therefore, a block expression can always be assigned to a lvalue. A field access and an object access are translated to *Core-Java*, by the rules [LVal−1] and [LVal−2]. The last two rule in Figure 4 translate a complex assignment that contains a binary operator.

Inner Class

We also translate inner classes into top-level classes. An inner class allows an object to export other objects with direct access to its own methods and instance fields. This mechanism increases the complexity of the program analyses. Therefore, it is desired to simplify an inner class to a normal class. Our translation uses a similar idea as [9]: (1) Rename all occurrences of inner class name to "outer$inner"; (2) Add a new field, "this$outer$inner", to inner class. This field points to the direct outer class enclosing the inner class; (3) Add a new formal parameter "this$outer$inner" to all inner class constructors and assign to the field introduced before; (4) Rename all outer class's fields and methods, which appear in the inner class to "this$outer$inner.fields" and "this$outer$inner.methods", respectively. Names "outer" and "inner" denote the names of the outer class and of the inner class, respectively. A similar algorithm is used for anonymous classes and local class declarations.

Array

Core-Java treats Java array type as a class. This allows subsequent program analyses to use any kind of subtyping for *Core-Java* array class. Note that Java assumes array type to be covariant. To illustrate the array treatment, we consider the following simple program:

```
//Java code
int [] x = {10,20};
x[1] = x[2];...
//Core-Java code
Arr_1_Int x; int y;
x.setSize(2);x.update(1,10);x.update(2,20);
y= x.sub(2);x.update(1,y) ...
```

Core-Java array class, `Arr_1_Int` is defined as follows:

```
class Arr_1_Int extends Object {
int size;
void setSize(int s) {}
void getSize() {}
void update(int i, int v){}
int sub(int i) {} }
```

Our translator can translate almost all Java features into *Core-Java* language. Its limitations are related to labels, `assert` statement, Java annotations and `enum` declaration. However, these features are not frequently used in real applications. Java genericity is discussed in the next section.

4 Support for Program Analyses

The goal of designing *Core-Java* language was to help the program analyses on the Java-like languages, especially those analyses which are type-based and modular. This section discusses some of the *Core-Java* features that support object-oriented program analyses.

Core-Java consists of a reduced number of syntactic forms as compared to the Java language. This simplifies the complexity of the abstract syntax tree, but also the rules of the static and dynamic semantics. The result is a program analysis with a simpler implementation and proof.

Although, *Core-Java* is a simpler language, it can encode almost all of the Java features. *Core-Java* constructs can be regarded as a subset of Java constructs. The only difference is the *Core-Java* block that can be used on the right hand side of an assignment. This kind of assignment is not valid in Java. However, a *Core-Java* program can easily be translated into a valid Java program, as is illustrated by the following example:

```
//valid Core-Java code, but invalid Java code
int x; ...  x= {int y; y=y+1;y}
//valid Java and Core-Java code
int x; ...  {int y; y=y+1;x=y}
```

Core-Java tries to stay close to Java, so that a program analysis conclusion about a *Core-Java* program can be mapped back to the Java source program. An additional support is given by the translator, that inserts information about the Java source location in the abstract syntax tree of the translated *Core-Java* program. Thus, an error reported by a typechecker at a line of *Core-Java* program can be mapped back to the corresponding Java source line.

Core-Java separates class declarations from the code. Thus, field initializers are part of the constructor, while the inner classes are translated into top-level classes. This separation helps the modular program analyses to be carried out at a per method level.

Core-Java in essence is an expression-oriented language, that makes easier the formulation of static and dynamic semantics. Expression languages are more suitable for type-based analyses that work on an abstract syntax tree rather than on a control flow graph. However, the syntax of *Core-Java* in Figure 1 contains some constructions that control the program flow. These constructions can be classified in the following categories:

- intra-method flow: that is not altered by exceptions or by `return`. However, this flow can be controlled by the `while` loop, `if`..`then`..`else`, `break` (forward jump) and `continue` (backward jump).
- return flow: that is given by the `return`
- exceptional flow: controlled by `throws` and `try`..`catch`

Another Java construction that can change the flow is `switch`, but this statement is directly translated into *Core-Java* as is illustrated by the following example:

```
switch (v) {
case 1:  a=1;break;
case 2:  a=2;break; ...}
```

and the corresponding *Core-Java* code is:

```
boolean b1;boolean b2;boolean v1;boolean v2;
v1=Primitives.eq(v,1);
if v1 then { a=1;b1=false};
v2 = Primitives.eq(v,2);
v2 = Primitives.or(v2,b1);
if v2 then { a=2;b2=false};...  }
```

To manage the different categories of flow, our type-based analyses [5, 4] use a tuple of types, *(intra-method type, return type, exceptional type)* similar with [6] to represent the type of an expression:

$$\Gamma \vdash e : t_n \# t_r \# t_a, \varphi$$

where t_n is the *intra-method type* that characterizes normal execution of the expression, t_r is the *return type*, denoting the type of a possible return expression from e, and t_a is the *exceptional type* that characterizes the exceptional execution of e.

A flow-insensitive analysis is not affected by `while` loop, `break` and `continue`. Flow-insensitive results can be composed in any order. In a sequence, the analysis result of the first subexpression does not affect the analysis of the second subexpression. On the other hand, flow-sensitive results cannot be composed in any order. The analysis result for a subexpression depends on the analysis result of the previous subexpressions. Thus, the flow is very important and any constructions that alter the flow have to be made explicit to support flow-sensitive analyses. Therefore, we further translate *Core-Java*. The `while` loops are converted to equivalent tail-recursive methods. To mimic the effects of loops, such converted methods differ from user-defined methods in that they use pass-by-reference parameters instead of pass-by-value ones. This conversion is for analysis purposes only; the while form is still used for execution. The following example illustrates the additional translation of `while` loop and of `break`, `continue` and `return` statements present inside the loop body.

```
while (i < 8) {
if a { break;}
if b { return i;}
if c {continue;}
i++;}
```

The `break` statement is transalted into a `return` expression, while the original `return` statement (that was inside the loop) is translated into a `throw` expression. Then, the loop is translated into a tail-recursive method, having as parameters all free variables of the original loop. The `continue` statement is translated into a `return` of the recursive call value. In order to recover the flow of the original `return` (that was inside the loop) method call site is placed inside a `try`..`catch` expression:

```
boolean cond;
cond = Primitives.lessThan(i,8);
try { this.meth(cond,i,a,b,c)}
catch (MyIntException excp)
{ int j; j = excp.val; return j }
```

New method, *meth* is defined as follows.

```
void meth(boolean cond,int i,boolean a,boolean b,boolean c)
        throws MyIntException {
if cond then {
        if a then { return };
        if b then { MyIntException ret;
         ret = new MyIntException(i);throw ret};
        if c then {this.meth(cond,i,a,b,c);return};
        i=Primitives.inc(i,1);
        cond = Primitives.lessThan(i,8);
        this.meth(cond,i,a,b,c)
}else{}}
```

class `MyIntException` is defined as follows:

```
class MyIntException extends Exception {
int val; void MyIntException(int s) {val=s} }
```

The algorithm also works for nested loops.

Core-Java can easily be extended with type annotations either to represent the output of a program analysis (e.g. the region annotations produced by the region inference [5]) or to support the translation of a type-annotated Java program (e.g. region annotations given by the programmer as input for the region typechecker [5]). In general, the type annotations consist of annotations attached to the normal Java types, invariants attached to the class/interface declarations, and pre and post conditions attached to the method declarations. The methods may also be changed to parametric methods that contain type variables as arguments. All these annotations are specific to each type-based program analysis. In order to support the translation of a type annotated Java program, the translator has to be modified to work with the new annotated types instead of the normal Java types. We experimented the passing of type annotations through translator in [4], where a Java program was annotated with variant parametric types.

5 Experimental Evaluation

We have constructed a prototype for our translator and we used it to help two analyses: a region inference for Java [5]) and a typechecker of a variant parametric type system for Java [4]. The translator has been implemented in Java and Haskell. It uses the Java reflection mechanism to recover the class types.

Programs	Java (KB)	Core-Java (KB)	Time (sec)
DB	178	197	1.9
Jess	546	1741	49
JLex	485	1313	52.2
junit	448	825	33.9
TelnetD	646	1138	7.5
VPoker	466	1030	18.7
BH	25	95	1.6
Bisort	8	23	0.7
Em3d	12	35	0.7
Health	12	50	0.8
MST	10	34	0.8
Perimeter	21	52	1.3
power	19	90	1
Treeadd	4	16	0.5
TSP	13	39	0.7
Voronoi	21	80	2.7

Figure 5. Statistics

We ran our translator on some medium and large program benchmarks (up to 10000 lines of code) and the results are shown in Figure 5. Experiments were performed on AMD X2 3800 with 2GB RAM running SUSE Linux 10.0. The first goal of our experiments was to verify the correctness of our translator. We ran the original Java program and the translated *Core-Java* program on the same inputs and we obtained the same results. As we mentioned in the previous section, a *Core-Java* program can easily be translated into a valid Java program. The second goal of the experiments was to check the scalability of our translation rules. Figure 5 shows the translation time and the size of the Java and *Core-Java* code for each program benchmark. The translation time is reasonable, but for some benchmark the size of the translated code was larger than we expected. We identified two main reasons for the code expansion: the first reason was the translation rule of `switch` statement and the second one was the introduction of temporary variables for constant values. Both problems were fixed in the current formalism presented in this paper.

6 Conclusion

Our translator is very useful in extending the experiments of our projects [5, 4] to real-world applications. Using the translation to *Core-Java*, we avoided the re-implementation of the original prototypes for the whole Java language.

However, the current translator uses the Java reflection mechanism to recover the class types and, therefore builds only a part of the global dependency graph. Type-based analyses require a global dependency graph to analyse the program in a modular way.

Our intension is to have an integrated framework: translator, global dependency graph and any other specific data structure that can help and simplify the program analyses. We are currently working on a new prototype, fully implemented in Haskell. Haskell provides superior handling of symbolic data structures which makes it easier to implement complex translation rules. Another aspect is the correctness of the translation rules. We experimentally validated that translated programs are correct and currently we are working on a formal proof of the translation rules.

References

[1] D. Ancona, F. Damiani, S. Drossopoulou, and E. Zucca. Polymorphic bytecode: Compositional compilation for java-like languages. In *ACM POPL*. ACM Press, 2005.

[2] D. Ancona and G. Lagorio. Stronger typings for separate compilation of Java-like languages. In *FTfJP*, 2003.

[3] G. Bierman, M. Parkinson, and A. Pitts. MJ: An imperative core calculus for Java and Java with effects. Technical report, Cambridge University, April 2003.

[4] W.-N. Chin, F. Craciun, S.-C. Khoo, and C. Popeea. A Flow-Based Approach for Variant Parametric Types. In *ACM OOPSLA*, Portland, Oct. 2006. to appear.

[5] W.-N. Chin, F. Craciun, S. Qin, and M. Rinard. Region Inference for an Object-Oriented Language. In *ACM PLDI*, pages 243–254, Washington, DC, June 2004. ACM Press.

[6] S. Drossopoulou, T. Valkevych, and S. Eisenbach. Java type soundness revisited. Tech report, Imperial College, 1999.

[7] C. Flanagan and S. N. Freund. Type-based race detection for Java. In *ACM PLDI*. ACM Press, June 2000.

[8] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java(TM) Language Specification, Third Edition*. Addison-Wesley, 2005.

[9] A. Igarashi and B. Pierce. On inner classes. In *ECOOP*, pages 129–153, Cannes, France, June 2000. Springer.

[10] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A Minimal Core Calculus for Java and GJ. In *ACM OOPSLA*, Denver, Colorado, Nov. 1999. ACM Press.

[11] G. Necula, S. McPeak, S. P. Rahul, and W. Weimer. Cil: Intermediate language and tools for analysis and transformation of C programs. In *CC'02*, France, March 2002.