Core-Java: An Expression-Oriented Java

Florin Craciun Hong Yaw Goh Corneliu Popeea Wei-Ngan Chin

Department of Computer Science, National University of Singapore

{craciunm,gohhy,corneliu,chinwn}@comp.nus.edu.sg

Abstract

A common practice for 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. Such a lightweight fragment lacks important Java features, thus the 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. We propose an intermediate solution through *Core-Java*, an expression-oriented core calculus of 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.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Object-oriented languages

General Terms Design, Languages

Keywords language design, type-based analysis

1. Introduction

We propose a framework for rapid 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 the pure functional calculus, Featherweight Java [8], but it incorporates imperative features as Middleweight Java [1], and concurrent features as the small multithreaded calculus, Concurrent Java [6]. In contrast to the main motivations of these proposals, we are interested in a more practical core calculus that incorporates all Java features [7] and makes it easy to analyze and manipulate real-world 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.

Our *Core-Java* language is more abstract than typical intermediate languages designed for compilation as these intermediate languages lose structural information about types, loops and other high-level constructs.

OOPSLA'06 October 22–26, 2006, Portland, Oregon, USA.
ACM 1-59593-491-X/06/0010.

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. Additionally, *Core-Java* language contains constructions that control the program flow, as well as *imperative* and *object-oriented* features.

2. Core-Java Language

A program P (Figure 1) consists of a set of class and interface declarations, def. The language supports simple inheritance by extending a class, but also multiple inheritance through the interface mechanism.

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

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

Figure 1. The Syntax of *Core-Java*

A class contains the declared fields and methods while an interface contains only methods without body. A method declaration $meth$ consists of the method modifiers, its return type, method name, its arguments with their types, its raised exceptions, 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#. A block expression *eb* of the form $\{(\tau\ v)^* e\}$ consists of a list of local variable declarations and a body expression e. 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 expression blocks, assignments, cast oper-

ations, method invocations, conditionals, control-flow primitives, loops, exceptions, and concurrency 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*. 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. Java array type is treated as a class. *Core-Java* does not have operators on primitive types, instead it defines a special class, called *Primitives* that contains a static method for each primitive operator.

3. Java to Core-Java Translation

We formulate the translation as a set of type-directed rules that follow 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 type. Our algorithm consists of three main steps:

Generating Descriptors of Compilation Units. For each compilation unit, we generate its attached descriptor file. 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.

Computing the 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). 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.

Translation. The type-based translation is formulated as a modular type inference for the Java input program. The type system behind translation is similar with that formalised in [5]. The main judgement has the following form: $D, \Gamma \vdash e \Rightarrow_{exp} e' : \tau$ 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. Details about our translation rules can be found in the companion technical report [4].

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. *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* contains some constructions that control the program flow. These constructions can be classified in the following categories: (1) intra-method flow: that is not altered by exceptions or by `return`, but it can be controlled by the `while` loop, `if..then..else`, `break` (forward jump) and `continue` (backward jump); (2) return flow: that is given by the `return`; and (3) exceptional flow: controlled by `throws` and `try..catch`. To manage the different categories of flow, we use a tuple of types, *(intra-method type, return type, exceptional type)* similar with [5] 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. On the other hand, for the flow-sensitive analyses the program flow is important and the constructions that alter it complicate the formulation of such analyses. Therefore, we further translate *Core-Java*: `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.

Core-Java was designed to support annotations to represent either the output of a program analysis (e.g. the region annotations produced by the region inference [3]) or the translation of a Java program with type based annotations (e.g. region annotations given by the programmer as input for the region typechecker [3]). In general, the analyses use three kinds of annotations: annotations of the normal Java types, invariants for the class declarations, and pre/post conditions attached to the method declarations. All these annotations are specific to each type-based program analysis. In order to support a specific analysis, the translator has to be specialised to work with the new annotated types instead of the normal Java types. We experimented the passing of type annotations through translator in [2], where a Java program was annotated with variant parametric types. *Core-Java* methods were extended to parametric methods that contain type variables as arguments.

5. Conclusion and Future Work

We have implemented the translator in Haskell and we used it to help two analyses: a region inference for Java [3] and a typechecker of a variant parametric type system for Java [2]. The translator was very useful in extending the experiments of our projects to real-world applications.

Our goal is to have an integrated framework: translator, global dependency graph and any other specific data structure that can help and simplify the program analyses. 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.

Acknowledgements

This work is supported by AStar* research grant R-252-000-233-305.

References

[1] Gavin Bierman, Matthew Parkinson, and Andrew Pitts. MJ: An imperative core calculus for Java and Java with effects. Technical report, Cambridge University, 2003.

[2] Wei-Ngan Chin, Florin Craciun, Siau-Cheng Khoo, and Corneliu Popeea. A Flow-Based Approach for Variant Parametric Types. In *ACM OOPSLA*, Portland, 2006.

[3] Wei-Ngan Chin, Florin Craciun, Shengchao Qin, and Martin Rinard. Region Inference for an Object-Oriented Language. In *ACM PLDI*, Washington, 2004.

[4] Florin Craciun, Hong Yaw Goh, and Wei-Ngan Chin. A Framework for Object-Oriented Program Analyses via Core-Java. Technical report, National University of Singapore, 2006. avail. at ⟨http://www.comp.nus.edu.sg/~chinwn/papers/corejava.ps⟩.

[5] Sophia Drossopoulou, Tanya Valkevych, and Susan Eisenbach. Java type soundness revisited. Tech report, Imperial College, 1999.

[6] Cormac Flanagan and Stephen N. Freund. Type-based race detection for Java. In *ACM PLDI*. ACM Press, 2000.

[7] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, Third Edition*. Addison-Wesley, 2005.

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