Declare Your Language

Eelco Visser

Dynamic Languages Symposium
October 27, 2015
The Java™ Language Specification
Java SE 7 Edition

James Gosling
Bill Joy
Guy Steele
Glad Bracha
Alex Buckley

2012-07-27

Describing the Semantics of Java and Proving Type Soundness

Sophia Demopoulos and Simon Elebash
Department of Computing
Imperial College of Science, Technology and Medicine

1 Introduction

Java combines the experience from the development of several object oriented languages, such as C++, Smalltalk and Oz. The philosophy of the language designers was to include only features with already known semantics, and to provide a small and simple language.

Nevertheless, we feel that the introduction of some new features in Java, as well as the specific combination of features, justifies a study of the Java formal semantics. The use of interfaces, reminiscent of Smalltalk, is a simplification of the signature extension for C++ but, to the best of our knowledge, novel. The mechanism for dynamic method binding is that of C++, but we know of no formal definition. Java adopts the Smalltalk approach whereby all object variables are implicitly pointers.

Furthermore, although there are a large number of studies of the semantics of isolated programming language features or of minimal programming languages [35, 59, 66], there have not been many studies of the formal semantics of actual programming languages. In addition, the interplay of features which are very well understood in isolation, might introduce unexpected effects.
The Spoofax Language Workbench

Spoofax is a platform for developing textual domain-specific languages with full-featured Eclipse editor plugins.

With the Spoofax language workbench, you can write the grammar of your language using the high-level SDF grammar formalism. Based on this grammar, basic editor services such as syntax highlighting and code folding are automatically provided. Using high-level descriptor languages, these services can be customized. More sophisticated services such as error marking and content completion can be specified using rewrite rules in the Stratego language.

Meta Languages

Language definitions in Spoofax are constructed using the following meta-languages:

- The SDF3 syntax definition formalism
- The NaBL name binding language
- The TS type specification language
- The Stratego transformation language

[Kats, Visser. OOPSLA 2010]
Language Engineering

Syntax Checker
Name Resolver
Type Checker
Code Generator

The Java™ Language Specification
Java SE 7 Edition
James Gosling
Bill Joy
Guy Steele
Glad Bracha
Alex Buckley

Describing the Semantics of Java and Proving Type Soundness
Sophia Demopoulos and Simon Eisenbach
Department of Computing
Imperial College of Science, Technology and Medicine

1. Introduction

Java combines the experience from the development of several object-oriented languages such as C++, Smalltalk and C. The philosophy of the language designers was to include only features with already known semantics, to provide a small and simple language.

Nevertheless, we feel that the introduction of some new features in Java, as well as the specific combination of features, justify a study of the Java formal semantics. The use of interfaces, reminiscent of Smalltalk, is a simplification of the signature extension for C++. It is to the best of our knowledge, novel. The mechanism for dynamic method binding is that of C++, but we have no formal definition. Java adopts the Smalltalk approach whereby all object variables are implicitly pointers.

Furthermore, although there are a large number of studies of the semantics of isolated language features or of minimal programming languages, there have not been many studies of the formal semantics of actual programming languages. In addition, the interplay of features which are very well understood in isolation, might introduce unexpected effects.
Language Design

Syntax Definition  Name Binding  Type Constraints  Dynamic Semantics  Transform

A Language Designer’s Workbench

[Visser, Tilmach, Wachsmuth et al.; Onward 2014]
A Theory of Name Resolution

Language Design

Syntax Definition
Name Binding
Type Constraints
Dynamic Semantics
Transform

A Language Designer’s Workbench

[Neron, Tolmach, Visser, Wachsmuth et al.; ESOP 2015]
Declarative Syntax Definition

SDF2, SDF3 [Visser and many others 1994-2015]
Declare Your Syntax

[Kats, Visser, Wachsmuth; Onward 2010]
Language = Set of Sentences

fun (x : Int) { x + 1 }

text is a convenient interface for writing and reading programs
Language = Set of Trees

tree is a convenient interface for transforming programs
Tree Transformation

tree is a convenient interface for transforming programs

- **Syntactic**
  - coloring
  - outline view
  - completion

- **Semantic**
  - transform
  - translate
  - eval
  - analyze
  - refactor
  - type check
Tree Transformation

Mul(Int("3"),
  Add(VarRefId("x"),
      VarRefId("y"))))

→
Add(Mul(Int("3"),
         VarRefId("x")),
    Mul(Int("3"),
        VarRefId("y"))))

Mul(e1, Add(e2, e3)) → Add(Mul(e1, e2), Mul(e1, e3))
Language = Sentences \textit{and} Trees

different representations convenient for different purposes
From Text to Tree and Back

\[ 3 \cdot (x + y) \]

\[ (3 \cdot x) + (3 \cdot y) \]

parse

format

transform
SDF3 defines Trees and Sentences

\[
\begin{align*}
\text{Expr.Int} &= \text{INT} \\
\text{Expr.Add} &= \langle\langle \text{Expr} \rangle + \langle\langle \text{Expr} \rangle \rangle \\
\text{Expr.Mul} &= \langle\langle \text{Expr} \rangle \ast \langle\langle \text{Expr} \rangle \rangle \\
\end{align*}
\]

format (tree to text) + trees (structure) => parse (text to tree)

\[
\text{parse}(s) = t \quad \text{where} \quad \text{format}(t) \equiv s \quad \text{(modulo layout)}
\]
Grammar Engineering in Spoofax
Ambiguity
Ambiguity

\[ 3 \times x + y \]

\( t_1 \neq t_2 \land \text{format}(t_1) = \text{format}(t_2) \)
Declarative Disambiguation

Disambiguation Filters [Klint & Visser; 1994], [Van den Brand, Scheerder, Vinju, Visser; CC 2002]
Priority and Associativity

context-free syntax
Expr.Int = INT
Expr.Add = \(<\text{Expr}\) + \(<\text{Expr}\)\) \{left\}
Expr.Mul = \(<\text{Expr}\) * \(<\text{Expr}\)\) \{left\}

context-free priorities
Expr.Mul > Expr.Add

Recent improvement: safe disambiguation of operator precedence [Afroozeh et al. SLE13, Onward15]
Multi-Purpose Declarative Syntax Definition

Syntax Definition

\[ \text{Exp.Ifz} = < \]
\[ \text{ifz} <\text{Exp}> \text{ then} \]
\[ <\text{Exp}> \]
\[ \text{else} \]
\[ <\text{Exp}> \]
\[ > \]

Parser

Error recovery rules

Pretty-Printer

Abstract syntax tree

Syntactic coloring

Syntactic completion

Folding rules

Outline rules
Declare Your Syntax: Summary

(1) language-specific grammar + disambiguation rules

(2) language-independent spec of well-formed trees for grammar

(3) formatting based on layout hints in grammar

(4) parser generated automatically

(4’) no need to understand parsing algorithm

(4”) debugging in terms of representation

(5) syntactic and semantic operations abstract from parsing
Declare Your Names
A Theory of Name Resolution

Pierre Neron\textsuperscript{1} \hspace{1cm} Andrew Tolmach\textsuperscript{2} \hspace{1cm} Eelco Visser\textsuperscript{1} \hspace{1cm} Guido Wachsmuth

\textsuperscript{1}TU Delft \hspace{2cm} \textsuperscript{2}Portland State University

[ESOP15]
Language = Set of Graphs

module A {
  def s = 5
}

module B {
  import A
  def x = 6
  def y = 3 + s
  def f =
    fun x { x + y }
}
Name Resolution is Pervasive

Appears in many different artifacts...

... with rules encoded in many different ad-hoc ways

No standard approach, no re-use
Contrast with Syntax

A unique definition

A standard formalism

Supports

Context-Free Grammars

program = decl*
decl = module id { decl* }
| import qid
| def id = exp
exp = qid
| fun id { exp }
| fix id { exp }
| let bind* in exp
| letrec bind* in exp
| letpar bind* in exp
| exp exp
| exp ⊕ exp
| int
qid = id
| id . qid
bind = id = exp

Parser

AST

Pretty-Printing

Highlighting
Representing Bound Programs

- Many approaches to representing the results of name resolution within an (extended) AST, e.g.
  - numeric indexing [deBruijn72]
  - higher-order abstract syntax [PfenningElliott88]
  - nominal logic approaches [GabbayPitts02]
- Good support for binding-sensitive AST manipulation
- But: Do not say how to resolve identifiers in the first place!
  - Also: Can’t represent ill-bound programs
  - And: Tend to be biased towards lambda-like bindings
Binding Specification Languages

• Many proposals for domain-specific languages (DSLs) for specifying binding structure of a (target) language, e.g.
  • Ott [Sewell+10]
  • Romeo [StansiferWand14]
  • Unbound [Weirich+11]
  • Caml [Pottier06]
  • NaBL [Konat+12]
• Generate code to do resolution and record results
The NaBL Name Binding Language
Multi-Purpose Name Binding Rules

module names

namespaces Variable

binding rules

Var(x) :
   refers to Variable x

Param(x, t) :
   defines Variable x of type t

Fun(p, e) :
   scopes Variable

Fix(p, e) :
   scopes Variable

Let(x, t, e1, e2) :
   defines Variable x of type t in e2
Binding Specification Languages

- Many proposals for domain-specific languages (DSLs) for specifying binding structure of a (target) language, e.g.
  - Ott [Sewell+10]
  - Romeo [StansiferWand14]
  - Unbound [Weirich+11]
  - Caml [Pottier06]
  - NaBL [Konat+12]
- Generate code to do resolution and record results
- But: what are the **semantics** of such a language?
The Missing Piece

• Answer: the meaning of a binding specification for language L should be given by a function from L programs to their "resolution structures"

• So we need a (uniform, language-independent) method for describing such resolution structures...

• ...that can be used to compute the resolution of each program identifier

• (or to verify that a claimed resolution is valid)
Design Goals

• Handle broad range of language binding features...
• ...using minimal number of constructs
• Make resolution structure language-independent
• Handle named collections of names (e.g. modules, classes, etc.) within the theory
• Allow description of programs with resolution errors
A Theory of Name Resolution

For *statically lexically scoped* languages

A unique representation

A standard formalism

Scope Graphs

Supports

Resolution

$\alpha$-equivalence

IDE Navigation

Refactoring tools

Reasoning tools
Resolution Scheme

Resolution of a reference in a scope graph:

Building a path from a reference node to a declaration node following path construction rules

*Parameterized by notions of path well-formedness and ordering
Scope Graphs by Example
**Simple Scopes**

```
def y1 = x2 + 1
def x1 = 5
```
Ambiguous Resolutions

\[
\begin{align*}
\text{def } & \boxed{x_1} = 5 & S0 \\
\text{def } & \boxed{x_2} = 3 \\
\text{def } & z_1 = x_3 + 1
\end{align*}
\]

\[
\begin{align*}
\text{match } t \text{ with} & \\
& \text{I A } \boxed{x} \text{ I B } \boxed{x} \Rightarrow \ldots
\end{align*}
\]
Lexical Scoping

Well formed path: $\text{R.P*.D}$
def \( x_3 = z_2 \) 5 7 \( S_0 \)

```
def \( z_1 = \)
    fun \( x_1 \) { \n        fun \( y_1 \) { \n            \( x_2 + y_2 \) \n        } \n    } \n```
module A₁ {
    def z₁ = 5
}

module B₁ {
    import A₂
    def x₁ = 1 + z₂
}

Imports

Associated scope

Import

Import Step
def \( z_3 = 2 \)

module \( A_1 \) {
    def \( z_1 = 5 \)
}

module \( B_1 \) {
    import \( A_2 \)
    def \( x_1 = 1 + z_2 \)
}

\[ I(\_).p' < P.p \]

\[ R.I(A_2).D < R.P.D \]
Imports vs. Includes

```
def z_3 = 2

module A_1 {
  def z_1 = 5
}

include A_2

def x_1 = 1 + z_2
```

```
import A_2

def x_1 = 1 + z_2
```

The diagram illustrates the relationships between the modules and the variables.

The implication:

\[ R.D < R.I(A_2).D \]
Qualified Names

module $N_1$
{
    def $s_1 = 5$
}

module $M_1$
{
    def $x_1 = 1 + N_2.s_2$
}
```
def s_1 = 5
module N_1 {
}
def x_1 = 1 + N_2.s_2
```

Well formed path: \texttt{R.P*.I(_)*.D}
Transitive vs. Non-Transitive

With transitive imports, a well formed path is \( R.P*.I(\_)*.D \)

With non-transitive imports, a well formed path is \( R.P*.I(\_)?.D \)
A Calculus for Name Resolution

Reachability

Well formed path: $R \cdot P^* \cdot \text{I}(_{-})^* \cdot D$

Visibility

$D < P.p$

$p < p'$

$s.p < s.p'$

$\text{I}(_{-}).p' < P.p$

$D < \text{I}(_{-}).p'$