Executing Declarative Language Definitions

Eelco Visser

TU Delft

MiniKanren 2020 | August 27, 2020
def fib(n: int) -> int:
    if n <= 1:
        return 1
    else:
        return fib(n-2) + fib(n-1)
```java
def fib(n, m):
    if n <= 1:
        return 1
    else:
        clm = fib(n-1) + fib(n)
        return clm

fib(6)
```

```
[08:48:06] ~/Desktop$ javac Fib.java
[08:48:10] ~/Desktop$ java Fib
Fib 6: 8
Fib 5: 8
```
```java
public class Fib {
    public static int calc(int n) {
        if (n < 2)
            return n;
        else
            return calc(n - 1) + calc(n - 2);
    }

    public static void main(String[] args)
        System.out.println("Fib 6: " + calc); System.out.println("Fib 5: " + calc);
}
```
The Java™ Language Specification

Java SE 7 Edition

James Gosling
Bill Joy
Guy Steele
Gilad Bracha
Alex Buckley

2012-07-27
```java
public class Fib {
    public static int calc(int n) {
        if (n < 2) return n;
        else return calc(n - 1) + calc(n - 2);
    }

    public static void main(String[] args) {
        System.out.println("Fib 6: ");
        calc(System.out.println("Fib 5: ");
    }
}
```
The Java™ Language Specification
Java SE 7 Edition

James Gosling
Bill Joy
Guy Steele
Glad Branch
Alex Buckley

2012-07-27

Describing the Semantics of Java and Proving Type Soundness
Sophia Drossopoulou and Susan 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 CLOS. The philosophy of the language design 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 C++, is a simplification of the signature semantics for C++ and is, 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 in addition, the interplay of features which are very well understood in isolation, might introduce unexpected effects.
The Java™ Language Specification
Java SE 7 Edition

James Gosling
Bill Joy
Guy Steele
Gilad Bracha
Alex Buckley

2012-07-27

Describing the Semantics of Java and Proving Type Soundness
Sophia Drossoupoli and Susan Ehrlich
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++ and Smalltalk, into one. The philosophy of the language designers was to include only features with already known semantics, and not 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 type inference [T] is a simplification of the type system in Java, which is to the best of our knowledge - novel. The mechanism for dynamic method binding in Java, but we know of no formal definition. Java adopts the Smalltalk [S] approach whereby all objects are implicitly pointers.

Furthermore, although there are a large number of studies of the semantics of object-oriented programming languages or of minimal programming languages [T, S], 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.

Syntax definition
Static semantics
Dynamic semantics

Abstract syntax
Type system
Operational semantics
Type soundness
Proof

Parser
Type checker
Code generator
Interpreter

Parser
Error recovery
Syntax highlighting
Outline
Code completion
Navigation
Type checker
Debugger
Language Design
The Java™ Language Specification
Java SE 7 Edition

James Gosling
Bill Joy
Guy Steele
Gilad Bracha
Alex Buckley

2012-07-27

Describing the Semantics of Java and Proving Type Soundness
Sophia Drossopoulou and Swen van den Berg
Department of Computer Science, Imperial College of Science, Technology and Medicine

1 Introduction

Java combines the experiences 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, 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, remainder of method definitions and a simplification of the signature extension for C++ are, to the best of our knowledge, novel.

The mechanisms for dynamic method finding are those 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, there have not been many studies of the formal semantics of actual programming languages. In addition, the interplay of features which are well understood in isolation, might introduce unexpected effects.
The Java™ Language Specification
Java SE 7 Edition

James Gosling
Bill Joy
Guy Steele
Gilaad Bracha
Alex Buckley

2012-07-27

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

1 Introduction
Java combines the experiences from the development of several object-oriented languages, such as C++, Smalltalk and CLOS. 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 Modula-3, is a simplification of the signature extension for C++ and is, 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 seman-
tics of isolated programming language features or of minimal programming lan-
guages, 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.
Multi-purpose Declarative Meta-Languages
A Recipe for Declarative Meta-Languages

Representation
- Language-independent representation

Specification Formalism
- Language-specific rules

Declarative Semantics
- Model (instance of representation) satisfies specification

Language-Independent Tools
- Reusable language-independent tools operate on representation
- Provide multiple interpretations
- Sound wrt declarative semantics
The Spoofax Language Workbench
A tool for implementing programming languages
- Open source and freely available
- Used in education, research, and industry
- Requires a lot of software engineering to maintain

A long term research project
- Incubator for language engineering research
- Basis for implementation and evaluation of 100+ papers
- Imperfect approximation of a language designer’s workbench
Spoofax History

Stratego/XT [1997-2005]
- Command-line language processors with SDF + Stratego (C/Linux)

Spoofax [2006-2011]
- Eclipse (IMP) IDEs from SDF + Stratego + ESV (Java)

Spoofax 2 [2012-2020]
- Spoofax-Core library with bindings for Eclipse, IntelliJ, command-line
- Meta-Languages: SDF3, Stratego, NaBL, NaBL2, Statix, DynSem, …

Spoofax 3 [2019-…]
- Live language development based on PIE build system
**Research**
- Language Engineering, Language Prototyping

**Education**
- Compiler Construction (MiniJava)

**Academic Workflow Engineering**
- WebDSL (researchr.org, WebLab, …)

**Industry**
- Oracle Labs: Graph Analytics
- Canon: Several DSLs
- Philips: Software Restructuring *(we’re hiring PhD students!)*
Meta-languages

- Syntax definition with SDF3
- Static semantics with Statix
- Data-flow analysis with FlowSpec
- Transformation with Stratego
- Dynamic Semantics with DynSem/Dynamix
- Editor service definition with ESV
Declarative Syntax Definition with SDF3
Declarative Syntax Definition

**Representation**
- Syntax trees

**Specification Formalism: SDF3**
- Productions + Constructors + Templates + Disambiguation

**Declarative Semantics**
- Well-formedness of syntax trees wrt syntax definition

**Language-Independent Tools**
- Parser
- Formatting based on layout hints in grammar
- Syntactic completion
module structure

imports Common

context-free start-symbols Exp

context-free syntax

Exp.Var = ID
Exp.Int = INT
Exp.Add = Exp "+" Exp
Exp.Fun = "function" "(" {ID ","}* ")" "{" Exp "}"
Exp.App = Exp "(" {Exp ","}* ")"
Exp.Let = "let" Bnd* "in" Exp "end"
Bnd.Bnd = ID "=" Exp

let
inc = function(x) { x + 1 }
in
inc(3)
end
let inc = function(x) { x + 1 } in
inc(3) end

let inc = function(x) { x + 1 } in
inc(3) end

let inc = function(x){
  x + 1
} in
inc(3) end

context-free syntax

Exp.Var = <<ID>>
Exp.Int = <<INT>>
Exp.Add = <<Exp> + <Exp>>
Exp.Fun = <function(<{ID ","}*>){
  <Exp>
}>
Exp.App = <<Exp>(<<Exp ","}*>>)
Exp.Let = <let
  <Bnd*> in
  <Exp>
> end
Bnd.Bnd = <<ID> = <Exp>>
Disambiguation
Ambiguity = Multiple Possible Parses

**context-free syntax**

- `Exp` = `<(Exp)> {bracket}
- `Exp.Int` = `INT`
- `Exp.Var` = `ID`
- `Exp.Add` = `<<Exp> + <Exp>>`
- `Exp.Fun` = `<function({ID ","})> <Exp>>`
- `Exp.App` = `<<Exp> <Exp>>`
- `Exp.Let` = `<let <Bnd>> in <Exp>>`
- `Bnd.Bnd` = `<<ID> = <Exp>>`
- `Exp.If` = `<if(<Exp>) <Exp>>`
- `Exp.IfElse` = `<if(<Exp>) <Exp> else <Exp>>`
- `Exp.Match` = `<match <Exp> with {{Case ","}}>>`
- `Case.Case` = `[[Pat] -> [Exp]]`
- `Pat.PVar` = `ID`
- `Pat.PApp` = `<<Pat> <Pat>>`

```
amb( [ Add(Var("a"), Add(Var("b"), Var("c")))
      , Add(Add(Var("a"), Var("b")), Var("c")))
    ]
  )
```
Disambiguation = Select(Structure)

context-free syntax

Exp = <((Exp)>){bracket}

Exp.Int = INT
Exp.Var = ID
Exp.Add = <<Exp> + <Exp>>

Exp.Fun = <function(<{ID ","}>>)<Exp>>
Exp.App = <<Exp> <Exp>>

Exp.Let = <let <Bnd*> in <Exp>>

Bnd.Bnd = <<ID> = <Exp>>

Exp.If = <if(<Exp>)<Exp>>
Exp.IfElse = <if(<Exp>)<Exp> else <Exp>>

Exp.Match = <match <Exp> with <{Case "|"}>>
Case.Case = [[Pat] → [Exp]]

Pat.PVar = ID
Pat.PApp = <<Pat> <Pat>>

\[ a + b + c \]

\[ \text{amb(} \]
\[ [ \text{Add(Var("a"), Add(Var("b"), Var("c")))} \]
\[ , \text{Add(Add(Var("a"), Var("b"), Var("c")))} ] \]
\[ \text{)} \]

\[ \text{Add(Add(Var("a"), Var("b"), Var("c"))} \]
context-free syntax

Exp = \langle \langle Exp \rangle \rangle \{bracket\}

Exp.Int = INT
Exp.Var = ID
Exp.Add = \langle\langle Exp \rangle + \langle Exp\rangle \{left\}

Exp.Fun = \langle function(\langle\{ID ","\}\rangle) \langle Exp\rangle\rangle
Exp.App = \langle\langle Exp \rangle \langle Exp\rangle \{left\}

Exp.Let = \langle let \langle Bnd\rangle in \langle Exp\rangle\rangle

Bnd.Bnd = \langle\langle ID \rangle = \langle Exp\rangle\rangle

Exp.If = \langle if\langle Exp\rangle \langle Exp\rangle\rangle
Exp.IfElse = \langle if\langle Exp\rangle \langle Exp\rangle else \langle Exp\rangle\rangle

Exp.Match = \langle match \langle Exp\rangle with \langle\{Case "]\}\rangle\rangle
\{longest-match\}
Case.Case = [[Pat] \rightarrow [Exp]]

Pat.PVar = ID
Pat.PApp = \langle\langle Pat \rangle \langle Pat\rangle \{left\}

context-free priorities

Exp.App > Exp.Add > Exp.IfElse > Exp.If
> Exp.Match > Exp.Let > Exp.Fun
Associativity = Solve Intra Operator Ambiguity

context-free syntax
Exp = <(<Exp>)> {bracket}
Exp.Int = INT
Exp.Var = ID
Exp.Add = <<Exp> + <Exp>> {left}
Exp.Fun = <function({ID ","}*) <Exp>>
Exp.App = <<Exp> <Exp>> {left}
Exp.Let = <let <Bnd*> in <Exp>>
Bnd.Bnd = <<ID> = <Exp>>
Exp.If = <if(<Exp>) <Exp>>
Exp.IfElse = <if(<Exp>) <Exp> else <Exp>>
Exp.Match = <match <Exp> with {{Case "|"}+>>
  {longest-match}
Case.Case = [[Pat] → [Exp]]
Pat.PVar = ID
Pat.PApp = <<Pat> <Pat>> {left}
context-free priorities
Exp.App > Exp.Add > Exp.IfElse > Exp.If
  > Exp.Match > Exp.Let > Exp.Fun

a + b + c

amb(
  [ Add(Var("a"), Add(Var("b"), Var("c"))),
    Add(Add(Var("a"), Var("b")), Var("c"))
  ]
)

Add(Add(Var("a"), Var("b")), Var("c"))
Priority = Solve Inter Operator Ambiguity

context-free syntax

Exp = <(<Exp>)> {bracket}

Exp.Int = INT
Exp.Var = ID
Exp.Add = <Exp> + <Exp> {left}

Exp.Fun = <function(<{ID ","}*> <Exp>) <Exp>
Exp.App = <Exp> <Exp> {left}

Exp.Let = <let <Bnd*> in <Exp>>
Bnd.Bnd = <ID> = <Exp>

Exp.If = <if(<Exp>) <Exp>>
Exp.IfElse = <if(<Exp>) <Exp> else <Exp>>

Exp.Match = <match <Exp> with <{Case "|"}+>>
  {longest-match}
Case.Case = [[Pat] → [Exp]]

Pat.PVar = ID
Pat.PApp = <Pat> <Pat> {left}

costext-free priorities

Exp.App > Exp.Add > Exp.IfElse > Exp.If
> Exp.Match > Exp.Let > Exp.Fun

f a + b

amb(
  [ Add(App(Var("f"), Var("a")), Var("b"))
  , App(Var("f"), Add(Var("a"), Var("b")))
  ]
)

Add(App(Var("f"), Var("a")), Var("b"))
Dangling Else = Operators with Overlapping Prefix

context-free syntax
Exp = \langle\langle Exp\rangle\rangle \{bracket\}
Exp.Int = INT
Exp.Var = ID
Exp.Add = \langle\langle Exp\rangle\rangle + \langle\langle Exp\rangle\rangle \{left\}
Exp.Fun = \langle\langle function(ID,\ldots)\rangle\rangle \langle\langle Exp\rangle\rangle
Exp.App = \langle\langle Exp\rangle\rangle \langle\langle Exp\rangle\rangle \{left\}
Exp.Let = \langle\langle let Bnd in Exp\rangle\rangle
Bnd.Bnd = \langle\langle ID \rangle\rangle = \langle\langle Exp\rangle\rangle
Exp.If = \langle\langle if(Exp)\rangle\rangle \langle\langle Exp\rangle\rangle
Exp.IfElse = \langle\langle if(Exp)\rangle\rangle \langle\langle Exp\rangle\rangle else \langle\langle Exp\rangle\rangle
Exp.Match = \langle\langle match Exp with \{Case \ldots\}+\rangle\rangle \{longest-match\}
Case.Case = [[Pat] \rightarrow [Exp]]
Pat.PVar = ID
Pat.PApp = \langle\langle Pat\rangle\rangle \langle\langle Pat\rangle\rangle \{left\}

context-free priorities
Exp.App > Exp.Add > Exp.IfElse > Exp.If
> Exp.Match > Exp.Let > Exp.Fun

if(1) if(2) 3 else 4

amb(
  [ IfElse(
      Int("1")
      , Int("2") , Int("3")
      , Int("4")
    )
    , If(
        Int("1")
        , IfElse(Int("2") , Int("3") , Int("4"))
    )
  ]
)

If(
  Int("1")
  , IfElse(Int("2") , Int("3") , Int("4"))
)
Parenthesize
Parenthesize = Disambiguate\(^{-1}\) (Insert Necessary Parentheses)

**context-free syntax**

- \(\text{Exp} = <\langle\text{Exp}\rangle>\) \(\{\text{bracket}\}\)
- \(\text{Exp.Int} = \text{INT}\)
- \(\text{Exp.Var} = \text{ID}\)
- \(\text{Exp.Add} = <<\text{Exp}>> + <\text{Exp}>>\) \(\{\text{left}\}\)
- \(\text{Exp.Fun} = <\text{function}(\langle\{\text{ID} ","\}\rangle) > <\text{Exp}>>\)
- \(\text{Exp.App} = <<\text{Exp}>> <\text{Exp}>>\) \(\{\text{left}\}\)
- \(\text{Exp.Let} = <\text{let} <\text{Bnd}\rangle in <\text{Exp}>>\)
- \(\text{Bnd.Bnd} = <<\text{ID}>> = <\text{Exp}>>\)
- \(\text{Exp.If} = <\text{if}(<\text{Exp}>) <\text{Exp}>>\)
- \(\text{Exp.IfElse} = <\text{if}(<\text{Exp}>) <\text{Exp}>> \text{else} <\text{Exp}>>\)
- \(\text{Exp.Match} = <\text{match} <\text{Exp}> with \langle\{\text{Case "|"}\}\rangle>>\)
  \(\{\text{longest-match}\}\)
- \(\text{Case.Case} = [[\text{Pat}] \rightarrow [\text{Exp}]]\)
- \(\text{Pat.PVar} = \text{ID}\)
- \(\text{Pat.PApp} = <<\text{Pat}>> <\text{Pat}>>\) \(\{\text{left}\}\)

**context-free priorities**

- \(\text{Exp.App} > \text{Exp.Add} > \text{Exp.IfElse} > \text{Exp.If} > \text{Exp.Match} > \text{Exp.Let} > \text{Exp.Fun}\)

- \((a + b) + c\)
- \(\text{Add(Add(Var("a"), Var("b")), Var("c"))}\)
- \(a + b + c\)
Parenthesize = Disambiguate$^{-1}$ (Insert Necessary Parentheses)

\[
a + (\text{let } x = b \text{ in } (c + d))
\]

**context-free syntax**

- $\text{Exp} = <(\text{Exp})> \ {\text{bracket}}$
- $\text{Exp.Int} = \text{INT}$
- $\text{Exp.Var} = \text{ID}$
- $\text{Exp.Add} = <<\text{Exp}> + <\text{Exp}>> \ {\text{left}}$
- $\text{Exp.Fun} = <\text{function}(\{\text{ID} ",\}\}) <\text{Exp}>>$
- $\text{Exp.App} = <<\text{Exp}>> <\text{Exp}>> \ {\text{left}}$
- $\text{Exp.Let} = <\text{let} <\text{Bnd}*>> \text{ in } <\text{Exp}>>$
- $\text{Bnd.Bnd} = <<\text{ID}>> <\text{Exp}>>$
- $\text{Exp.If} = <\text{if}(\text{Exp}) <\text{Exp}>>$
- $\text{Exp.IfElse} = <\text{if}(\text{Exp}) <\text{Exp}>> \text{ else } <\text{Exp}>>$
- $\text{Exp.Match} = <\text{match} \text{Exp} \text{ with } \{\text{Case } |\}>> \ {\text{longest-match}}$
- $\text{Case.Case} = [[[\text{Pat}] \rightarrow [\text{Exp}]$
- $\text{Pat.PVar} = \text{ID}$
- $\text{Pat.PApp} = <<\text{Pat}>> <\text{Pat}>> \ {\text{left}}$

**context-free priorities**

- $\text{Exp.App} > \text{Exp.Add} > \text{Exp.IfElse} > \text{Exp.If} > \text{Exp.Match} > \text{Exp.Let} > \text{Exp.Fun}$
Parenthesize = Disambiguate\(^{-1}\) (Insert Necessary Parentheses)

context-free syntax
Exp = \((\langle\text{Exp}\rangle)\) \{bracket\}

Exp.Int = INT
Exp.Var = ID
Exp.Add = \(\langle\text{Exp}\rangle + \langle\text{Exp}\rangle\) \{left\}

Exp.Fun = \(\langle\text{function}(\langle\text{ID} \"\"\rangle)*\rangle\) \langle\text{Exp}\rangle
Exp.App = \(\langle\text{Exp}\rangle \langle\text{Exp}\rangle\) \{left\}

Exp.Let = \(\langle\text{let}\langle\text{Bnd}\rangle*\rangle \text{ in} \langle\text{Exp}\rangle\)

Bnd.Bnd = \(\langle\text{ID}\rangle = \langle\text{Exp}\rangle\)

Exp.If = \(\langle\text{if}\langle\text{Exp}\rangle\rangle \langle\text{Exp}\rangle\)
Exp.IfElse = \(\langle\text{if}\langle\text{Exp}\rangle\rangle \langle\text{Exp}\rangle \text{ else} \langle\text{Exp}\rangle\)

Exp.Match = \(\langle\text{match}\langle\text{Exp}\rangle \text{ with} \langle\text{Case} \"\|\"\rangle\rangle\)
\{longest-match\}
Case.Case = \(\langle\text{Pat}\rangle \rightarrow \langle\text{Exp}\rangle\)

Pat.PVar = ID
Pat.PApp = \(\langle\text{Pat}\rangle \langle\text{Pat}\rangle\) \{left\}

case-free priorities
Exp.App > Exp.Add > Exp.IfElse > Exp.If > Exp.Match > Exp.Let > Exp.Fun

(a + (let x = b in c)) + d

Add(
Add(
   Var("a")
 , Let([Bnd("x", Var("b"))], Var("c"))
)
, Var("d")
)
a + (let x = b in c) + d
SDF3 Interpretations

Statement.If = <
  if(<Exp>)
  <Statement>
else
  <Statement>
>

- Parser
- Error recovery
- Pretty-printer
- Abstract syntax tree schema
- Syntactic coloring
- Syntactic completion
- Folding rules
- Outline rules
Generating Artifacts from Syntax Definitions

Grammar

ParseGen

Parse Table

Algebraic Signature

Completion Generator

Program

Parser

AST

AST

Completion Rules

Formatter Generator

Formatting Rules

Format Rules

Generated Artifact

User-Defined Specification

Language Independent Generator

User-Defined Specification

Generated Artifact
Declarative Type System Specification with Statix
Declarative Static Semantics Definition with Statix

Representation
- Scope graph

Specification Formalism: Statix
- Type constraints + scope graph constraints + resolution policies

Declarative Semantics
- Scope graph of program satisfies specification

Language-Independent Tools
- Type checking
- Refactoring / Renaming
- Code completion
module Library{
  module Sig {
    type Pred = Int → Bool
  }

  type P = Sig.Pred

  module Odd {
    import Even
    def odd : P =
      fun(x) if x == 0 then false else even (x - 1)
  }

  module Even {
    import Odd
    def even : Sig.Pred =
      fun(x) if x == 0 then true else odd (x - 1)
  }

  module Application {
    import Library.Even
    // def alias = Library // error

    $ even 42
    $ Library.Odd.odd 45
  }
}
Logic Programming
Statix is a pure logic programming language

A Statix specification defines predicates

If a predicate holds for some term, the term has the property represented by the predicate

- `typeOfExp(s, e) == T`  
  expression $e$ has type $T$ in scope $s$

- `typeOfType(s, t) == T`  
  syntactic type $t$ has semantic type $T$ in scope $s$

- `declOk(s, d)`  
  declaration $d$ is well-defined (Ok) in scope $s$

Statix is a pure logic programming language

A Statix specification defines predicates

If a predicate holds for some term, the term has the property represented by the predicate

- `typeOfExp(s, e) == T`  
  expression $e$ has type $T$ in scope $s$

- `typeOfType(s, t) == T`  
  syntactic type $t$ has semantic type $T$ in scope $s$

- `declOk(s, d)`  
  declaration $d$ is well-defined (Ok) in scope $s$
Functional Notation vs Predicate Notation

**Rules**

\[
\begin{align*}
typeOfExp : & \text{scope} \times \text{Exp} \rightarrow \text{TYPE} \\
typeOfType : & \text{scope} \times \text{Type} \rightarrow \text{TYPE}
\end{align*}
\]

**Example**

\[
\begin{align*}
typeOfExp(s, e) &= T \\
\text{expression } e \text{ has type } T \text{ in scope } s
\end{align*}
\]

One expression has one type

(Solver does not match on type argument)

**Rule**

\[
\begin{align*}
typeOfExp(s, e, T) &= T \\
\text{expression } e \text{ has type } T \text{ in scope } s
\end{align*}
\]

One expression can have multiple types
Predicates are Defined by Rules

**Predicate**

\[
\text{typeOfExp} : \text{scope} * \text{Type} \to \text{TYPE}
\]

**Rule**

\[
\text{typeOfExp}(s, \text{Add}(e_1, e_2)) = \text{INT}() :\-
\text{typeOfExp}(s, e_1) = \text{INT}(),
\text{typeOfExp}(s, e_2) = \text{INT}()
\]

For all \(s, e_1, e_2\)

If the premises are true, the head is true
From Declarative Definition to Type Checker

1> 1 + 2 * 3
2
3> true && false
4
5> 1 ^ 2
6
7> true + 4
8
9> 1 && (true || false)
10
11> if 1 == 1 then
12>      true
13>      else
14>      1 == 3
15
16> if 1 == 1 then
17>      true
18>      else
19>      2

Parser

Syntax Errors

Signature in Statix

Type System in Statix

Solver

AST

Syntax Highlighting
Programs with Names
module Names {

module Even {
    import Odd
    def even = fun(x)
        if x == 0 then true else odd(x - 1)
}

module Odd {
    import Even
    def odd = fun(x)
        if x == 0 then false else even(x - 1)
}

module Compute {
    type Result = { input : Int, output : Bool }
    def compute = fun(x)
        Result{ input = x, output = Odd.odd x }
}
}

Name binding key in programming languages

Many name binding patterns

Deal with erroneous programs

Name resolution complicates type checkers, compilers

Ad hoc non-declarative treatment

A systematic, uniform approach to name resolution?
let function fact(n : int) : int =
    if n < 1 then
        1
    else
        n * fact(n - 1)
in
fact(10)

Name Resolution with Scope Graphs

Program

let function fact(n : int) : int =
    if n < 1 then
        1
    else
        n * fact(n - 1)
in
fact(10)

Scope Graph

Name Resolution
Declaring and Resolving Names
Declarations and References

<table>
<thead>
<tr>
<th>signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>constructors</td>
</tr>
<tr>
<td>Var : ID -&gt; Exp</td>
</tr>
<tr>
<td>Def : Bind -&gt; Decl</td>
</tr>
<tr>
<td>Bind : ID * Exp -&gt; Bind</td>
</tr>
<tr>
<td>rules</td>
</tr>
<tr>
<td>typeOfExp(s, Var(x)) = typeOfVar(s, x).</td>
</tr>
<tr>
<td>declOk(s, Def(bind)) :-</td>
</tr>
<tr>
<td>bindOk(s, s, bind).</td>
</tr>
<tr>
<td>bindOk(s_bnd, s_ctx, Bind(x, e)) :- {T}</td>
</tr>
<tr>
<td>typeOfExp(s_ctx, e) == T,</td>
</tr>
<tr>
<td>declareVar(s_bnd, x, T).</td>
</tr>
</tbody>
</table>

| def a = 0 |
| def b = a + 1 |
| def c = a + b |

> a + b + c
rules

declareVar : scope * string * TYPE

typeOfVar : scope * string → TYPE

de  a = 0
def b = a + 1
def c = a + b
> a + b + c

declaration and reference
Representing Name Binding with Scope Graphs

**signature**

namespaces

Var : string

name-resolution

resolve Var filter e

relations

typeOfDecl : occurrence $\rightarrow$ TYPE

**rules**

declareVar : scope $\times$ string $\times$ TYPE

type0fVar : scope $\times$ string $\rightarrow$ TYPE

def a = 0
def b = a + 1
def c = a + b
> a + b + c
### Signature

**Namespaces**

- `Var : string`

**Name-resolution**

- `resolve Var filter e`

**Relations**

- `typeOfDecl : occurrence -> TYPE`

### Rules

- `declareVar : scope * string * TYPE`
- `typeOfVar : scope * string -> TYPE`

**Declaration and Reference**

- `declareVar(s, x, T) :-
  s -> Var{x} with typeOfDecl T.`

### Checkpoint

- `def a = 0`
- `def b = a + 1`
- `def c = a + b`
- `> a + b + c`
Representing Name Binding with Scope Graphs

**signature**

namespaces

Var : string

name-resolution

resolve Var filter e

relations

typeOfDecl : occurrence $\rightarrow$ TYPE

**rules**

declareVar : scope $\times$ string $\times$ TYPE
typeOfVar : scope $\times$ string $\rightarrow$ TYPE

\[
declareVar(s, x, T) :- \\
s \rightarrow \text{Var}\{x\} \text{ with } \text{typeOfDecl} \ T.
\]

\[
typeOfVar(s, x) = T :- \{x'\} \\
typeOfDecl \text{ of } \text{Var}\{x\} \text{ in } s \leftrightarrow \{\_ , (\text{Var}\{x'\}, T)\}.
\]

---

def a = 0
def b = a + 1
def c = a + b
> a + b + c

---

variable x is declared in scope s with type T

variable x in scope s resolves to declaration x' with type T
How about shadowing?
Lexical Scope
New Scope and Scope Edge Constraints

**signature**

**constructor**

```
Let : ID * Exp * Exp -> Exp
```

**rules**

```
typeOfExp(s, Let(x, e1, e2)) = T :- {S s_let}
  typeOfExp(s, e1) = S,
  new s_let,
  s_let -P-> s,
  declareVar(s_let, x, S),
  typeOfExp(s_let, e2) = T.
```
Path Wellformedness: Reachability

**signature**
- constructors
  - Let : ID * Exp * Exp \to Exp

**rules**
- \( \text{typeOfExp}(s, \text{Let}(x, e1, e2)) = T \defeq \{S \ s_let\} \)
- \( \text{typeOfExp}(s, e1) = S, \)
- \( \text{new} \ s_let, \)
- \( s_let -P\to s, \)
- \( \text{declareVar}(s_let, x, S), \)
- \( \text{typeOfExp}(s_let, e2) = T. \)

**signature**
- namespaces
  - Var : string
- name-resolution
  - resolve Var \textit{filter} P*

**path** \( P* \) allows resolution through zero or more \( P \) edges

```
let a = 1 in
let b = 2 in
let c = 3 in
a + b + c
```
Path Specificity: Visibility (Shadowing)

**signature**

```
constructors
  Let : ID * Exp * Exp -> Exp
```

**rules**

```
typeOfExp(s, Let(x, e1, e2)) = T :- \{S s_let\}
  typeOfExp(s, e1) = S,
  new s_let,
  s_let -P-> s,
  declareVar(s_let, x, S),
  typeOfExp(s_let, e2) = T.
```

**signature**

```
namespaces
  Var : string
name-resolution
  resolve Var filter P* min $ < P
```

`path P* allows resolution through zero or more P edges`

`prefer local scope ($) over parent scope (P)`
How about non-lexical bindings?
Non-Lexical Scope (Modules)
### Modules: Scopes as Types

**signature**
- **constructors**
  - MOD : scope $\rightarrow$ TYPE
  - Module : ID * list(Decl) $\rightarrow$ Decl
  - Import : ID $\rightarrow$ Decl

**rules**
- declOk(s, Module(m, decls)) :- \{ s_mod \}
  - new s_mod, s_mod -P s,
  - declareMod(s, m, MOD(s_mod)),
  - declsOk(s_mod, decls).

**signature**
- **namespaces**
  - Mod : string

**example code**
```python
def c = 0
module A {
    import B
    def a = b + c
}
module B {
    def b = 2
}
```

**diagram**
- The diagram illustrates the lexical scope and the scope as type concept, with modules A and B and their associated variables c, a, and b.
Resolving Import

**Signature**

**Constructors**

- `MOD : scope -> TYPE`
- `Module : ID * list(Decl) -> Decl`
- `Import : ID -> Decl`

**Rules**

- `declOk(s, Module(m, decls)) :- {s_mod}
  new s_mod, s_mod -P-> s,
  declareMod(s, m, MOD(s_mod)),
  declsOk(s_mod, decls).`

- `declOk(s, Import(p)) :- {s_mod s_end}
  typeOfModRef(s, p) == MOD(s_mod),
  s -I-> s_mod.`

**Signature**

**Namespaces**

- `Mod : string`

**Name-resolution**

- `resolve Mod`
- `filter Px`
- `min $ < I, $ < P, I < P`
**signature**

**constructors**

MOD : scope → TYPE  
Module : ID * list(Decl) → Decl  
Import : ID → Decl

**rules**

dclOk(s, Module(m, decls)) :- {s_mod}
  new s_mod, s_mod -P→ s, 
  declareMod(s, m, MOD(s_mod)), 
  dclOk(s_mod, decls).

dclOk(s, Import(p)) :- {s_mod s_end}
  typeOfModRef(s, p) MOD(s_mod), 
  s -I→ s_mod.

**signature**

**namespaces**

Mod : string

name-resolution

resolve Mod
  filter P*
  min $ < I, $ < P, I < P

---

```python
def c = 0
module A {
  import B
  def a = b + c
}
module B {
  def b = 2
}
```

---

```
signature
c: Int
rules
dclOk(s, Module(m, decls)) :- {s_mod}
  new s_mod, s_mod -P→ s, 
  declareMod(s, m, MOD(s_mod)), 
  dclOk(s_mod, decls).
dclOk(s, Import(p)) :- {s_mod s_end}
  typeOfModRef(s, p) MOD(s_mod), 
  s -I→ s_mod.

---

```

---

```python
def c = 0
module A {
  import B
  def a = b + c
}
module B {
  def b = 2
}
```
**Resolving through Import Edge**

### Signature
- **Constructors**
  - `MOD : scope -> TYPE`
  - `Module : ID * list(Decl) -> Decl`
  - `Import : ID -> Decl`

### Rules
- `declOk(s, Module(m, decls)) :- {s_mod}`
  - `new s_mod, s_mod -> s,`
  - `declareMod(s, m, MOD(s_mod)),`
  - `declsOk(s_mod, decls).`
- `declOk(s, Import(p)) :- {s_mod s_end}`
  - `typeOfModRef(s, p) == MOD(s_mod),`
  - `s -> I -> s_mod.`

### Signature
- **Namespaces**
  - `Var : string`
- **Name-resolution**
  - `resolve Var`
  - `filter P * I *`
  - `min $ < I, $ < P, I < P`

---

```python
def c = 0
module A {
    import B
    def a = b + c
}
module B {
    def b = 2
}
```
signature constructors
MOD : scope -> TYPE
Module : ID * list(Decl) -> Decl
Import : ID -> Decl

rules
def b = 0
module A {
  import B
  def a = b
}
module B {
  def b = 2
}

def b = 0
module A {
  import B
  def a = b
}
module B {
  def b = 2
}

signature namespaces
Var : string
name-resolution
resolve Var
  filter P * I*
  min $ < I, $ < P, I < P

prefer blue path over red path
prefer import
resolve through import edges
lexical scope
scope as type
scope as type
resolve import
import edge
**Mutual Imports**

**signature**

```
signature
constructors
  MOD  : scope → TYPE
  Module : ID * list(Decl) → Decl
  Import : ID → Decl
```

**rules**

```
def c = 0
module A {
  import B
  def a = b + c
}
module B {
  import A
  def b = 2
  def d = a + c
}
```

**signature**

```
signature
namespaces
  Var : string
name-resolution
  resolve Var
    filter P* I*
    min $ < I, $ < P, I < P
```
Mutual Imports

signature
  constructors
  MOD : scope → TYPE
  Module : ID * list(Decl) → Decl
  Import : ID → Decl

rules

declOk(s, Module(m, decls)) :- {s_mod}
  new s_mod, s_mod -P→ s,
  declareMod(s, m, MOD(s_mod)),
  declsOk(s_mod, decls).

declOk(s, Import(p)) :- {s_mod s_end}
  typeOfModRef(s, p) == MOD(s_mod),
  s -I→ s_mod.

signature
  namespaces
  Var : string
  name-resolution
  resolve Var
    filter P* I*
    min $ < I, $ < P, I < P
Transitive Import

**signature**

**constructors**

 MOD : scope → TYPE
 Module : ID * list(Decl) → Decl
 Import : ID → Decl

**rules**

\(\text{declOk}(s, \text{Module}(m, \text{decls})) :- \{s\text{\_mod}\}\)

new \(s\text{\_mod}\), \(s\text{\_mod} \rightarrow s\),

declareMod(s, m, MOD(s\text{\_mod})),
declsOk(s\text{\_mod}, \text{decls}).

\(\text{declOk}(s, \text{Import}(p)) :- \{s\text{\_mod} \text{\_end}\}\)

typeOfModRef(s, p) \text{MOD}(s\text{\_mod}),

\(s \rightarrow I \rightarrow s\text{\_mod}\).

**signature**

**namespaces**

Var : string
name-resolution
resolve Var
filter \(P \times I\)
min \(\$ < I, \$ < P, I < P\)
**signature**

**constructors**

- MOD : \text{scope} \rightarrow \text{TYPE}
- Module : \text{ID} * \text{list(Decl)} \rightarrow \text{Decl}
- Import : \text{ID} \rightarrow \text{Decl}

**rules**

\[
\text{declOk}(s, \text{Module}(m, \text{decls})) :\{-s\text{mod}\}
\]
- new $s\text{mod}$, $s\text{mod} -P\rightarrow s$,
- declareMod($s$, $m$, MOD($s\text{mod}$)),
- declsOk($s\text{mod}$, decls).

\[
\text{declOk}(s, \text{Import}(p)) :\{-s\text{mod s\text{end}}\}
\]
- typeOfModRef($s$, $p$) \equiv \text{MOD}($s\text{mod}$),
- $s -I\rightarrow s\text{mod}$.

**signature**

**namespaces**

- Var : \text{string}
- name-resolution
- resolve Var
  - filter $P * I *$
  - min $\$ < I, \$ < P, I < P$
Statix Interpretations
Declarative Semantics [OOPSLA’18]
- $G \models \text{programOk}(s, p)$
- Does program $p$ satisfy the programOk predicate in scope $s$, given scope graph $G$?

Type Checking
- Given a program term $p$, what is valid scope graph $G$?
- Operational semantics is safe wrt declarative semantics [OOPSLA’20]
- Type check programs concurrently and/or incrementally

Code Completion [ECOOP’19]
- Given a hole (placeholder) in an incomplete program, what are valid completions?

Renaming
- Given a name $x$ in a program, can it be renamed to $y$, without being captured?

Quick Fixes
- Given a name/type error in a program, what is repair that would solve the error?

Random Term Generation
- Given a placeholder (and type), randomly generate a program that is syntactically, binding, and type correct

Statix Interpretations (In Progress)
Conclusion
Multi-purpose Declarative Meta-Languages
Multi-purpose Declarative Meta-Languages
More Information

More Information

The Spoofax Language Workbench

Specif is a platform for developing textual domain-specific programming languages. The platform provides the following ingredients:

- Meta-languages for high-level declarative language definition
- An interactive environment for developing languages using these meta-languages
- Code generators that produces parsers, type-checkers, compilers, interpreters, and other tools from language definitions
- Generation of full-featured Eclipse editor plugins from language definitions
- Generation of full-featured Eclipse editor plugins from language definitions (behaviors)
- An API for programmatically combining the components of a language implementation

With Specif you can focus on the essence of language definition and ignore irrelevant implementation details.

Developing Software Languages

Specif supports the development of textual languages, but does not otherwise restrict what kind of language you develop. Specif has been used to develop the following kinds of languages:

- Programming languages
- Languages for generating code
- Domain-specific languages
- Languages that capture the understanding of a domain with linguistic abstractions. Design a DSL for your domain with a compiler that generates code that would be tedious and error-prone to produce manually.
- Tailoring languages
- Languages with a special run-time environment and interpreter
- Work-flow languages
- Languages for scheduling actions as building the components of a software system
- Configuration languages
- Languages for configuring software and other systems
- Data description languages
- Languages for formatting data
- Data modeling languages
- Languages for describing data schemas

http://eelcovisser.org

http://metaborg.org