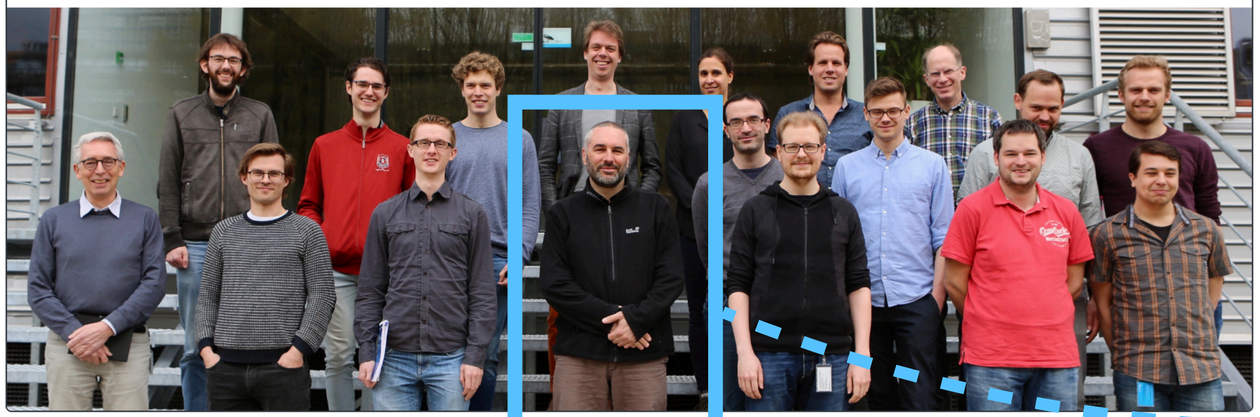
What is  Research?

ct

the effectiveness and reliability of programming languages and systems.

TUDelft PL Organization Education Research Events Contact

TU Delft Programming Languages



Eelco Visser

DOI

11:00 - 12:30: OOPSLA - Modular Verification at Olympia

Chair(s): Friedrich Steimann Fernuni Hagen

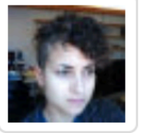
11:00 - 11:22 *Research paper*
☆ Modular Verification of Heap Reachability Properties in Separation Logic
Arshavir Ter-Gabrielyan ETH Zurich, Alexander J. Summers ETH Zurich, Peter Müller ETH Zurich
DOI Pre-print

11:22 - 11:45 *Talk*
☆ Modular Verification of Web Page Layout
Pavel Panchekha University of Utah, Michael D. Ernst University of Washington, USA, Zachary Tatlock University of Washington, Shoaib Kamil Adobe
DOI

11:45 - 12:07 *Talk*
☆ Modular Verification for Almost-Sure Termination of Probabilistic Programs
Mingzhang Huang Shanghai Jiao Tong University, Hongfei Fu Shanghai Jiao Tong University, Krishnendu Chatterjee IST Austria, Amir Kafshdar Goharshady IST Austria

Review Committee

Eelco Visser Chair
Delft University of Technology
Netherlands



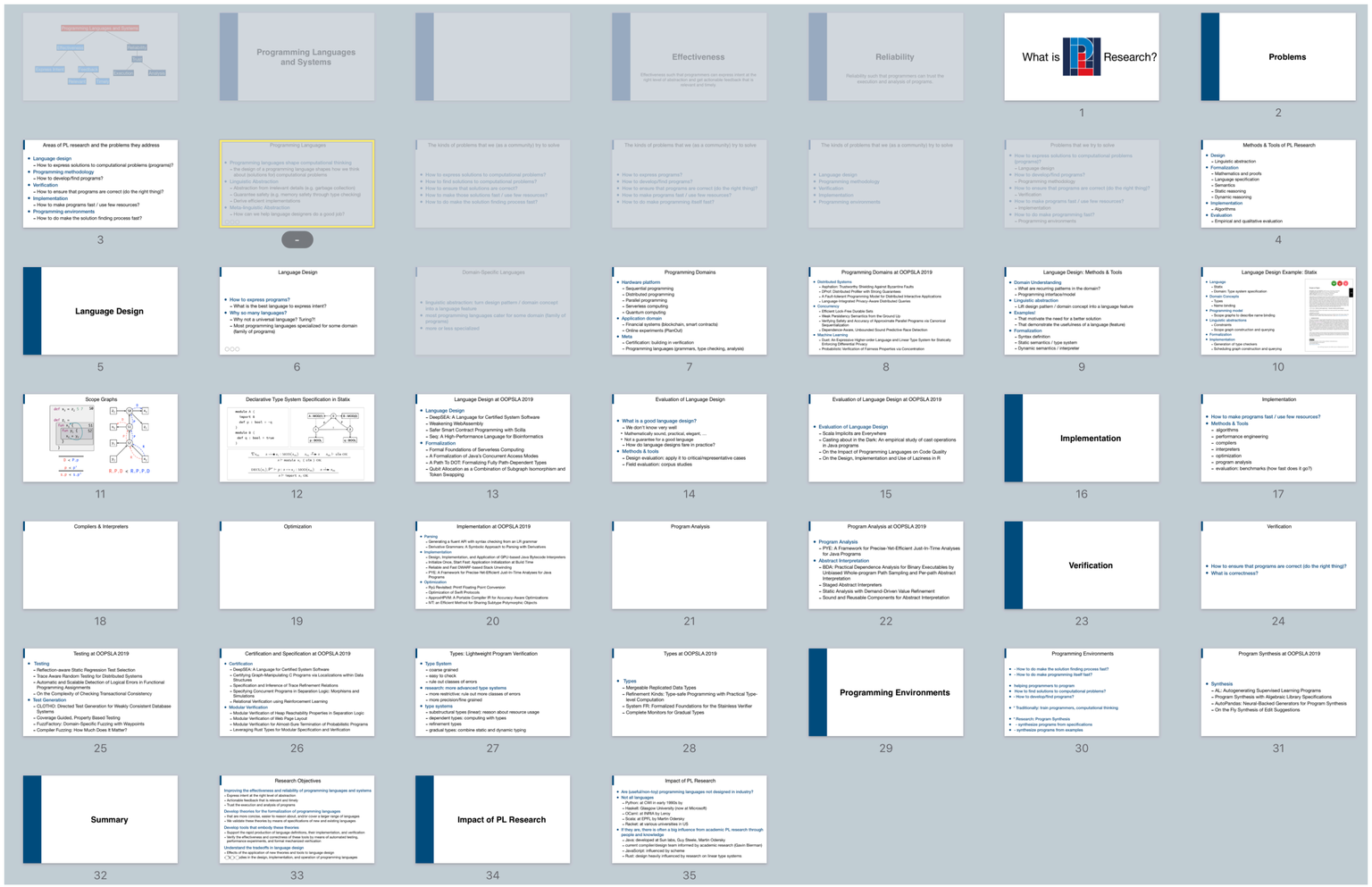
Sara Achour
MIT
United States

Nada Amin
Harvard University
United States

“We want to have a talk that introduces them to the kind of research that our community does without picking a single paper (or even topic) to discuss.”

— Karim, Marianna & Jonathan

Failed Attempt: A Taxonomy of PL Research



What is the essence of PL research?

Grand A Theory of PL Research

Eelco Visser

PL research is about getting stuff for free

What stuff?

PL research is about
getting ***programming stuff*** for free

What stuff?

PL research is about
getting ***programming stuff*** for free

Code

Performance Functionality Reliability

Free as in …

PL research is about
getting ***programming stuff*** for free

for free?

Free as in …

If you provide X, you get Y for free

Free as in …

If you provide X, you get Y for free

you = the programmer

get for free = by the programmer

the giving is done by an algorithm or theory

(which required (a lot of hard) work by PL researchers)

Free as in …

If you provide X, you get Y for free

For example

X = a program in a high-level programming language

Y = a program in machine code

Getting = compiling

(aka automatic programming)

Free as in …

If you provide X, you get Y for free

`Providing X’ done using
a (programming) language

Captures the properties
for which Y can be got

A recipe for PL research

Y takes a lot of effort

is tedious, slow, buggy, …

A recipe for PL research

Y takes a lot of effort

is tedious, slow, buggy, …

I wonder if there is some way to get Y for free?

A recipe for more incremental PL research (That is fine)

Some existing (F: X => Y)

Is there an X’ that takes less effort?

Can we extend Y?

Can we make F more efficient?

etc.

Understanding PL research

Many papers are about the intricacies of some F: X => Y

monadic strength mediates between the current computation and …

to make parsing incremental we use the old tree as input and break down …

That is the nitty-gritty that defines
the everyday life of a PL researcher

This theory provides a lens to understand PL research:

What is the goal they are trying to achieve? vs
What is the method they are using to do that?

What does PL research involve? What do PL researchers do?

Domain

- How does programming in this area work?

Design

- Description of the idea
  - Sketches, examples, prototypes, …

Specification

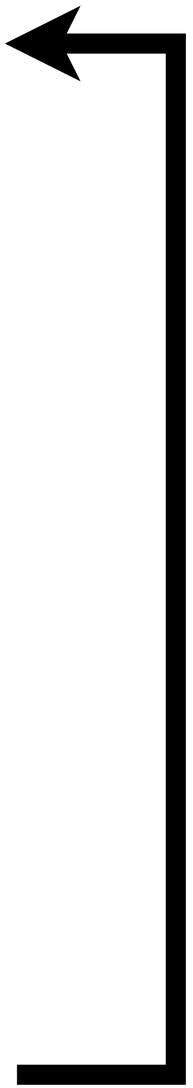
- Formal definition and verification
  - Data model, theorems, proofs

Implementation

- Realization as a software system
  - Algorithms, (performance) engineering, testing, deployment

Evaluation

- Does the approach achieve its purpose?
  - Performance benchmarking, coverage testing, user studies, …



Each of these areas have there own field of study

Theory in Practice

Example problems and solutions based on 8 papers [2010 - 2018]

How each solution unlocks new research problems

Programming Environments for Free

Problem

- Implementation of IDEs for programming languages is expensive

Provide

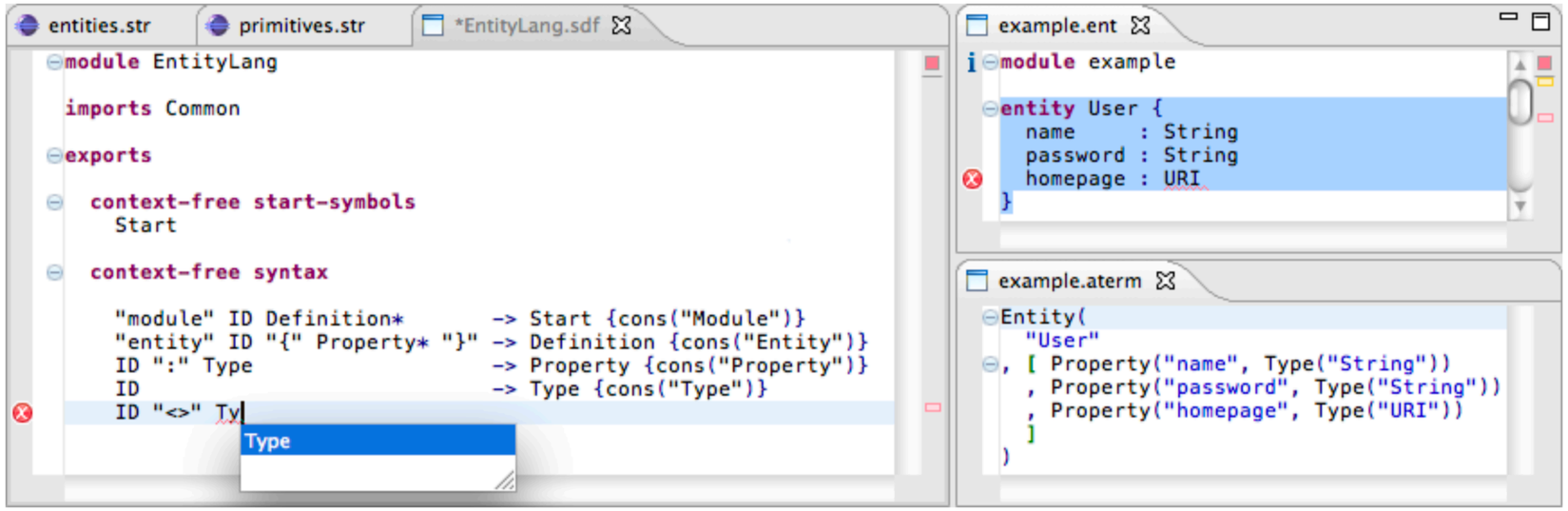
- Syntax definition, type checker, transformations for language

Get

- An IDE for language with syntactic and semantic editor services

Solution

- IDE engineering (e.g. dynamic language loading in Eclipse)



The Spoofax Language Workbench

Rules for Declarative Specification of Languages and IDEs

Lennart C. L. Kats
Delft University of Technology
l.c.l.kats@tudelft.nl

Eelco Visser
Delft University of Technology
visser@acm.org

Abstract

Spoofax is a language workbench for efficient, agile development of textual domain-specific languages with state-of-the-art IDE support. Spoofax integrates language processing techniques for parser generation, meta-programming, and IDE development into a single environment. It uses concise, declarative specifications for languages and IDE services. In this paper we describe the architecture of Spoofax and introduce idioms for high-level specifications of language semantics using rewrite rules, showing how analyses can be reused for transformations, code generation, and editor services such as error marking, reference resolving, and content completion. The implementation of these services is supported by language-parametric editor service classes that can be dynamically loaded by the Eclipse IDE, allowing new languages to be developed and used side-by-side in the same Eclipse environment.

Categories and Subject Descriptors D.2.3 [*Software Engineering*]: Coding Tools and Techniques; D.2.6 [*Software Engineering*]: Programming Environments

General Terms Languages

1. Introduction

Domain-specific languages (DSLs) provide high expressive power focused on a particular problem domain [38, 47]. They provide linguistic abstractions over common tasks within a domain, so that developers can concentrate on application logic rather than the accidental complexity of low-level implementation details. DSLs have a concise, domain-specific notation for common tasks in a domain, and allow reasoning at the level of these constructs. This allows them to be used for automated, domain-specific analysis, verification, optimization, parallelization, and transformation (AVOPT) [38].

OOPSLA/SPLASH'10, October 17–21, 2010, Reno/Tahoe, Nevada, USA.

For developers to be productive with DSLs, good integrated development environments (IDEs) for these languages are essential. Over the past four decades, IDEs have slowly risen from novelty tool status to becoming a fundamental part of software engineering. In early 2001, IntelliJ IDEA [42] revolutionized the IDE landscape [17] with an IDE for the Java language that parsed files as they were typed (with error recovery in case of syntax errors), performed semantic analysis in the background, and provided code navigation with a live view of the program outline, references to declarations of identifiers, content completion proposals as programmers were typing, and the ability to transform the program based on the abstract representation (refactorings). The now prominent Eclipse platform, and soon after, Visual Studio, quickly adopted these same features. No longer would programmers be satisfied with code editors that provided basic syntax highlighting and a "build" button. For new languages to become a success, state-of-the-art IDE support is now mandatory. For the production of DSLs this requirement is a particular problem, since these languages are often developed with much fewer resources than general purpose languages.

There are five key ingredients for the construction of a new domain-specific language. (1) A parser for the *syntax* of the language. (2) Semantic *analysis* to validate DSL programs according to some set of constraints. (3) *Transformations* manipulate DSL programs and can convert a high-level, technology-independent DSL specification to a lower-level program. (4) A *code generator* that emits executable code. (5) Integration of the language into an *IDE*.

Traditionally, a lot of effort was required for each of these ingredients. However, there are now many tools that support the various aspects of DSL development. Parser generators can automatically create a parsers from a grammar. Modern parser generators can construct efficient parsers that can be used in an interactive environment, supporting error recovery in case of syntax-incorrect or incomplete programs. Meta-programming languages [3, 10, 12, 20, 35] and frameworks [39, 57] make it much easier to specify the semantics of a language. Tools and frameworks for IDE development such as IMP [7, 8] and TMF [56], simplify the implementation of IDE services. Other tools, such as the Synthesizer

Name Resolution for Free

Problem

- Ad hoc implementation of name binding rules (for different language constructs, in various artifacts)

Provide

- Declarative specification of name binding rules

Get

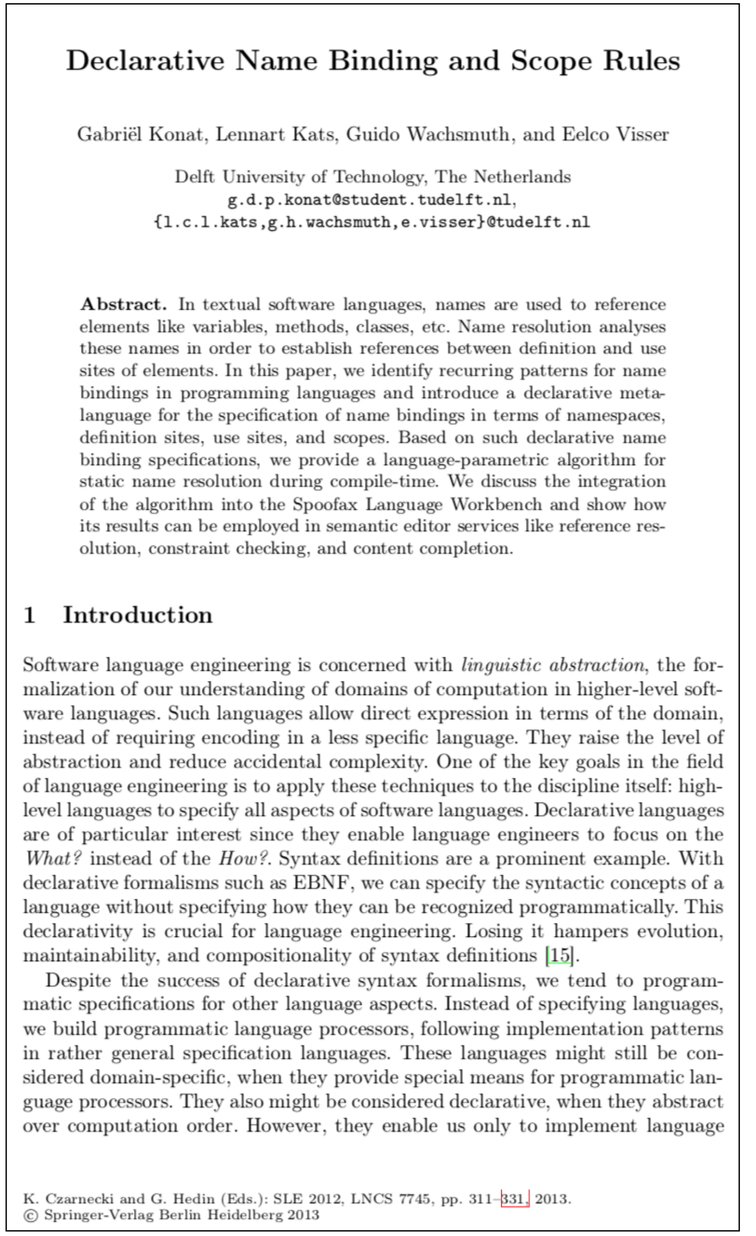
- A name analysis algorithm, name related editor services

Solution

- Name binding language design, name analysis algorithm, code generation

Evaluation

- Coverage evaluation: name binding for subset of C#



Declarative Name Binding and Scope Rules

Gabriël Konat, Lennart Kats, Guido Wachsmuth, and Eelco Visser

Delft University of Technology, The Netherlands
g.d.p.konat@student.tudelft.nl,
{l.c.l.kats,g.h.wachsmuth,e.visser}@tudelft.nl

Abstract. In textual software languages, names are used to reference elements like variables, methods, classes, etc. Name resolution analyses these names in order to establish references between definition and use sites of elements. In this paper, we identify recurring patterns for name bindings in programming languages and introduce a declarative meta-language for the specification of name bindings in terms of namespaces, definition sites, use sites, and scopes. Based on such declarative name binding specifications, we provide a language-parametric algorithm for static name resolution during compile-time. We discuss the integration of the algorithm into the Spoofax Language Workbench and show how its results can be employed in semantic editor services like reference resolution, constraint checking, and content completion.

1 Introduction

Software language engineering is concerned with *linguistic abstraction*, the formalization of our understanding of domains of computation in higher-level software languages. Such languages allow direct expression in terms of the domain, instead of requiring encoding in a less specific language. They raise the level of abstraction and reduce accidental complexity. One of the key goals in the field of language engineering is to apply these techniques to the discipline itself: high-level languages to specify all aspects of software languages. Declarative languages are of particular interest since they enable language engineers to focus on the *What?* instead of the *How?*. Syntax definitions are a prominent example. With declarative formalisms such as EBNF, we can specify the syntactic concepts of a language without specifying how they can be recognized programmatically. This declarativity is crucial for language engineering. Losing it hampers evolution, maintainability, and compositionality of syntax definitions [15].

Despite the success of declarative syntax formalisms, we tend to programmatic specifications for other language aspects. Instead of specifying languages, we build programmatic language processors, following implementation patterns in rather general specification languages. These languages might still be considered domain-specific, when they provide special means for programmatic language processors. They also might be considered declarative, when they abstract over computation order. However, they enable us only to implement language

K. Czarnecki and G. Hedin (Eds.): SLE 2012, LNCS 7745, pp. 311–331, 2013.

Name Resolution for Free

```
class C {
    void m() { int x; }
}

class D {
  void m() {
    int x;
    int y;
    { int x; x = y + 1; }
    x = y + 1;
  }
}
```

```
rules
  Class(NonPartial(), c, _, _):
    defines unique class c
    scopes field, method

  Class(Partial(), c, _, _):
    defines non-unique class c
    scopes field, method

  Method(_, m, _, _):
    defines unique method m
    scopes variable

  Block(_): scopes variable
```

```
namespaces class field method variable
rules
  Field(_, f)          : defines unique field f
  Method(_, m, _, _): defines unique method m
  Call(m, _)           : refers to method m

  Var(_, v): defines unique variable v
  VarRef(x): refers to variable x otherwise to field x
```

Declarative Name Binding and Scope Rules

Gabriël Konat, Lennart Kats, Guido Wachsmuth, and Eelco Visser

Delft University of Technology, The Netherlands
g.d.p.konat@student.tudelft.nl,
{l.c.l.kats,g.h.wachsmuth,e.visser}@tudelft.nl

Abstract. In textual software languages, names are used to reference elements like variables, methods, classes, etc. Name resolution analyses these names in order to establish references between definition and use sites of elements. In this paper, we identify recurring patterns for name bindings in programming languages and introduce a declarative meta-language for the specification of name bindings in terms of namespaces, definition sites, use sites, and scopes. Based on such declarative name binding specifications, we provide a language-parametric algorithm for static name resolution during compile-time. We discuss the integration of the algorithm into the Spoofax Language Workbench and show how its results can be employed in semantic editor services like reference resolution, constraint checking, and content completion.

1 Introduction

Software language engineering is concerned with *linguistic abstraction*, the formalization of our understanding of domains of computation in higher-level software languages. Such languages allow direct expression in terms of the domain, instead of requiring encoding in a less specific language. They raise the level of abstraction and reduce accidental complexity. One of the key goals in the field of language engineering is to apply these techniques to the discipline itself: high-level languages to specify all aspects of software languages. Declarative languages are of particular interest since they enable language engineers to focus on the *What?* instead of the *How?*. Syntax definitions are a prominent example. With declarative formalisms such as EBNF, we can specify the syntactic concepts of a language without specifying how they can be recognized programmatically. This declarativity is crucial for language engineering. Losing it hampers evolution, maintainability, and compositionality of syntax definitions [15].

Despite the success of declarative syntax formalisms, we tend to programmatic specifications for other language aspects. Instead of specifying languages, we build programmatic language processors, following implementation patterns in rather general specification languages. These languages might still be considered domain-specific, when they provide special means for programmatic language processors. They also might be considered declarative, when they abstract over computation order. However, they enable us only to implement language

K. Czarnecki and G. Hedin (Eds.): SLE 2012, LNCS 7745, pp. 311–331, 2013.

Incremental Analysis for Free

```
1  class User {
2    string name;
3  }
4  class Blog {
5    string post(User user, string message) {
6      posterName = "name";
7      string posterName;
8      posterName = user.nam;
9      string posterName = user.name;
10     return posterName;
11   }
12 }
```

Problem

- Whole program analysis is not suitable for interactive use

Provide

- (Same) Declarative specification of name binding rules

Get

- An ***incremental*** name analysis algorithm, name related editor services

Solution

- Algorithm and data structure design

Evaluation

- Performance benchmarking on series of git commits

A Language Independent Task Engine for Incremental Name and Type Analysis

Guido H. Wachsmuth[1,2], Gabriël D.P. Konat[1], Vlad A. Vergu[1], Danny M. Groenewegen[1], and Eelco Visser[1]

[1] Delft University of Technology, The Netherlands
{g.h.wachsmuth,v.a.vergu,d.m.groenewegen}@tudelft.nl,
{gkonat,visser}@acm.org
[2] Oracle Labs, Redwood City, CA, USA

Abstract. IDEs depend on incremental name and type analysis for responsive feedback for large projects. In this paper, we present a language-independent approach for incremental name and type analysis. Analysis consists of two phases. The first phase analyzes lexical scopes and binding instances and creates deferred analysis tasks. A task captures a single name resolution or type analysis step. Tasks might depend on other tasks and are evaluated in the second phase. Incrementality is supported on file and task level. When a file changes, only this file is recollected and only those tasks are reevaluated, which are affected by the changes in the collected data. The analysis does neither re-parse nor re-traverse unchanged files, even if they are affected by changes in other files. We implemented the approach as part of the Spoofax Language Workbench and evaluated it for the WebDSL web programming language.

1 Introduction

Integrated development environments (IDEs) provide a wide variety of language-specific editor services such as syntax highlighting, error marking, code navigation, content completion, and outline views in real-time, while a program is edited. These services require syntactic and semantic analyses of the edited program. Thereby, timely availability of analysis results is essential for IDE responsiveness. Whole-program analyses do not scale because the size of the program determines the performance of such analyses.

Incremental analysis reuses previous analysis results of unchanged program parts and reanalyses only parts affected by changes. The granularity of the incremental analysis directly impacts the performance of the analysis. A more fine-grained incremental analysis is able to reanalyze smaller units of change, but requires a more complex change and dependency analysis. At program level, any change requires reanalysis of the entire program, which might consider the results of the previous analysis. At file level, a file change requires reanalysis of the entire file and all dependent files. At program element level, changes to an element within a file require reanalysis of that element and dependent elements, but typically not of entire files. Incremental analyses are typically implemented

M. Erwig, R.F. Paige, and E. Van Wyk (Eds.): SLE 2013, LNCS 8225, pp. 260–280, 2013.

Incremental Analysis for Free

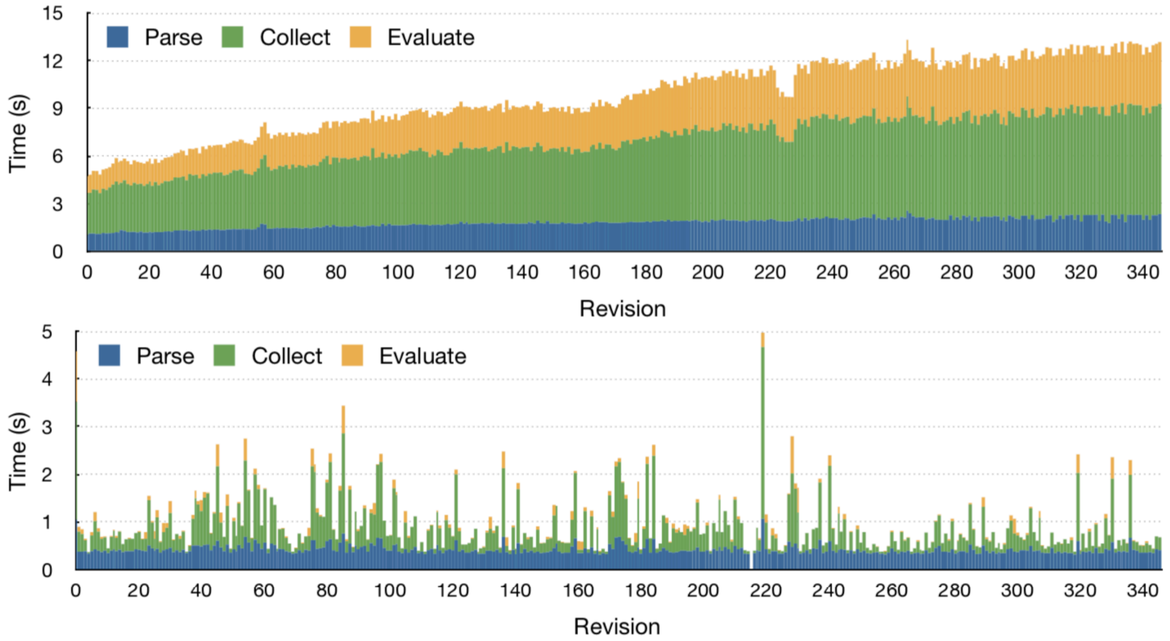


Fig. 8. Analysis time for full (top) and incremental (bottom) analyses

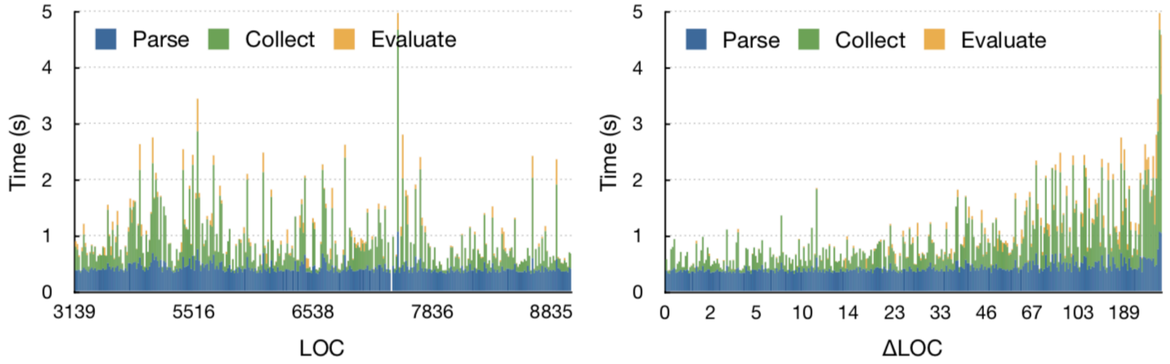


Fig. 9. Incremental analysis time ordered by LOC (left) and ΔLOC (right)

A Language Independent Task Engine for Incremental Name and Type Analysis

Guido H. Wachsmuth[1,2], Gabriël D.P. Konat[1], Vlad A. Vergu[1], Danny M. Groenewegen[1], and Eelco Visser[1]

[1] Delft University of Technology, The Netherlands
{g.h.wachsmuth,v.a.vergu,d.m.groenewegen}@tudelft.nl,
{gkonat,visser}@acm.org
[2] Oracle Labs, Redwood City, CA, USA

Abstract. IDEs depend on incremental name and type analysis for responsive feedback for large projects. In this paper, we present a language-independent approach for incremental name and type analysis. Analysis consists of two phases. The first phase analyzes lexical scopes and binding instances and creates deferred analysis tasks. A task captures a single name resolution or type analysis step. Tasks might depend on other tasks and are evaluated in the second phase. Incrementality is supported on file and task level. When a file changes, only this file is recollected and only those tasks are reevaluated, which are affected by the changes in the collected data. The analysis does neither re-parse nor re-traverse unchanged files, even if they are affected by changes in other files. We implemented the approach as part of the Spoofax Language Workbench and evaluated it for the WebDSL web programming language.

1 Introduction

Integrated development environments (IDEs) provide a wide variety of language-specific editor services such as syntax highlighting, error marking, code navigation, content completion, and outline views in real-time, while a program is edited. These services require syntactic and semantic analyses of the edited program. Thereby, timely availability of analysis results is essential for IDE responsiveness. Whole-program analyses do not scale because the size of the program determines the performance of such analyses.

Incremental analysis reuses previous analysis results of unchanged program parts and reanalyses only parts affected by changes. The granularity of the incremental analysis directly impacts the performance of the analysis. A more fine-grained incremental analysis is able to reanalyze smaller units of change, but requires a more complex change and dependency analysis. At program level, any change requires reanalysis of the entire program, which might consider the results of the previous analysis. At file level, a file change requires reanalysis of the entire file and all dependent files. At program element level, changes to an element within a file require reanalysis of that element and dependent elements, but typically not of entire files. Incremental analyses are typically implemented

M. Erwig, R.F. Paige, and E. Van Wyk (Eds.): SLE 2013, LNCS 8225, pp. 260–280, 2013.

More Name Resolution for Free

Problem

- But what is the semantics of those name binding rules?!
- And how to increase coverage?
  - e.g. how to express ‘subsequent scope’?

Provide

- Scope graph of a program + visibility policy

Get

- Resolution of references to declarations

Solution

- Paradigm: scope graphs
- Mathematical definition (calculus), name resolution algorithm, soundness proof

Evaluation

- Formalization of many typical binding patterns in programming languages

A Theory of Name Resolution

Pierre Neron[1], Andrew Tolmach[2], Eelco Visser[1], and Guido Wachsmuth[1]

[1)] Delft University of Technology, The Netherlands,
{p.j.m.neron, e.visser, g.wachsmuth}@tudelft.nl,
[2)] Portland State University, Portland, OR, USA
tolmach@pdx.edu

Abstract. We describe a language-independent theory for name binding and resolution, suitable for programming languages with complex scoping rules including both lexical scoping and modules. We formulate name resolution as a two-stage problem. First a language-independent scope graph is constructed using language-specific rules from an abstract syntax tree. Then references in the scope graph are resolved to corresponding declarations using a language-independent resolution process. We introduce a resolution calculus as a concise, declarative, and language-independent specification of name resolution. We develop a resolution algorithm that is sound and complete with respect to the calculus. Based on the resolution calculus we develop language-independent definitions of α-equivalence and rename refactoring. We illustrate the approach using a small example language with modules. In addition, we show how our approach provides a model for a range of name binding patterns in existing languages.

1 Introduction

Naming is a pervasive concern in the design and implementation of programming languages. Names identify *declarations* of program entities (variables, functions, types, modules, etc.) and allow these entities to be *referenced* from other parts of the program. Name *resolution* associates each reference to its intended declaration(s), according to the semantics of the language. Name resolution underlies most operations on languages and programs, including static checking, translation, mechanized description of semantics, and provision of editor services in IDEs. Resolution is often complicated, because it cuts across the local inductive structure of programs (as described by an abstract syntax tree). For example, the name introduced by a `let` node in an ML AST may be referenced by an arbitrarily distant child node. Languages with explicit name spaces lead to further complexity; for example, resolving a qualified reference in Java requires first resolving the class or package name to a context, and then resolving the member name within that context. But despite this diversity, it is intuitively clear that the basic concepts of resolution reappear in similar form across a broad range of lexically-scoped languages.

In practice, the name resolution rules of real programming languages are usually described using *ad hoc* and informal mechanisms. Even when a language *is* formalized, its resolution rules are typically encoded as part of static

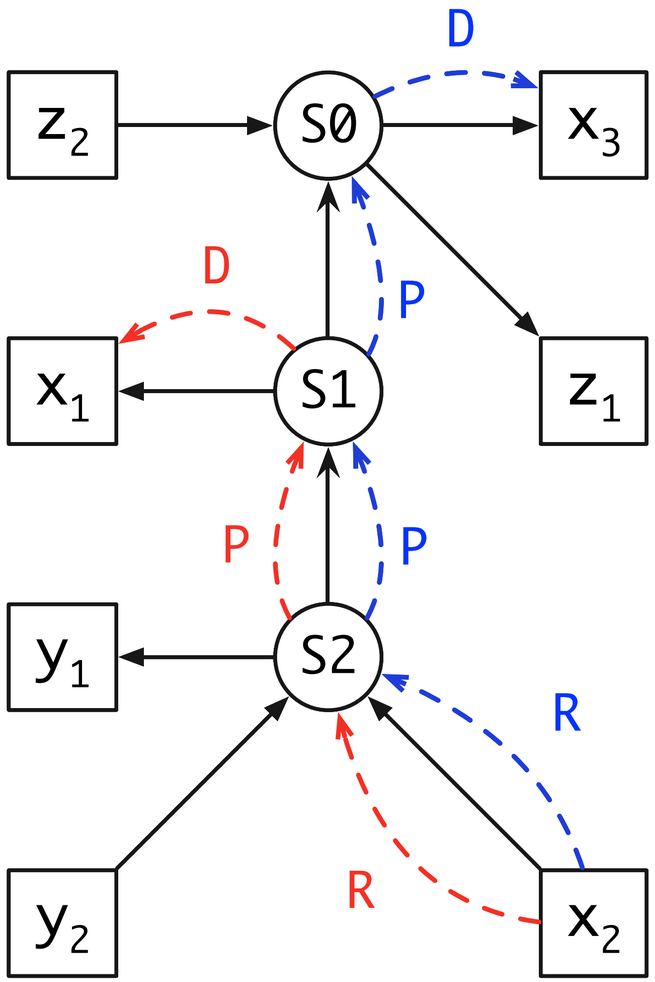
A scope graph represents the bindings of a program

```
def x3 = z2 5 7                    S0

def z1 =
  fun x1 {                         S1
    fun y1 {                       S2
      x2 + y2
    }
  }
```

$$\frac{}{D < P.p}$$

$$\frac{p < p'}{s.p < s.p'}$$



$$R.P.D < R.P.P.D$$

Type-Dependent Name Resolution for Free

Problem

- Type-dependent name resolution: name and type analysis are interdependent
- Program traversal of type checker depends on binding patterns of language

Provide

- Scope graph and type constraints for a program

Get

- Name and type resolution

Solution

- Represent bindings as constraints
- Formal definition of constraint language with sound resolution algorithm

Evaluation

- Constraint generation for LMR, language with representative binding patterns

A Constraint Language for Static Semantic Analysis Based on Scope Graphs

Hendrik van Antwerpen
TU Delft, The Netherlands
h.vanantwerpen@tudelft.nl

Pierre Néron
TU Delft, The Netherlands
p.j.m.neron@tudelft.nl

Andrew Tolmach
Portland State University, USA
tolmach@pdx.edu

Eelco Visser
TU Delft, The Netherlands
visser@acm.org

Guido Wachsmuth
TU Delft, The Netherlands
guwac@acm.org

Abstract

In previous work, we introduced *scope graphs* as a formalism for describing program binding structure and performing name resolution in an AST-independent way. In this paper, we show how to use scope graphs to build static semantic analyzers. We use *constraints* extracted from the AST to specify facts about binding, typing, and initialization. We treat name and type resolution as separate building blocks, but our approach can handle language constructs—such as record field access—for which binding and typing are mutually dependent. We also refine and extend our previous scope graph theory to address practical concerns including ambiguity checking and support for a wider range of scope relationships. We describe the details of constraint generation for a model language that illustrates many of the interesting static analysis issues associated with modules and records.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; D.3.2 [*Programming Languages*]: Language classifications; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs; D.3.4 [*Programming Languages*]: Processors; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages; D.2.6 [*Software Engineering*]: Programming Environments

Keywords Language Specification; Name Binding; Types; Domain Specific Languages; Meta-Theory

1. Introduction

Language workbenches [6] are tools that support the implementation of full-fledged programming environments for (domain-specific) programming languages. Ongoing research investigates how to reduce implementation effort by factoring out language-independent implementation concerns and providing high-level meta-languages for the specification of syntactic and semantic aspects of a language [18]. Such meta-languages should (i) have a clear and clean underlying theory; (ii) handle a broad range of common language features; (iii) be declarative, but be realizable by practical algorithms and tools; (iv) be factored into language-specific and language-independent parts, to maximize re-use; and (v) apply to erroneous programs as well as to correct ones.

In recent work we showed how name resolution for lexically-scoped languages can be formalized in a way that meets these criteria [14]. The name binding structure of a program is captured in a *scope graph* which records identifier declarations and references and their scoping relationships, while abstracting away program details. Its basic building blocks are *scopes*, which correspond to sets of program points that behave uniformly with respect to resolution. A scope contains identifier declarations and references, each tagged with its position in the original AST. Scopes can be connected by edges representing lexical nesting or import of named collections of declarations such as modules or records. A scope graph is constructed from the program AST using a language-dependent traversal, but thereafter, it can be processed in a largely language-independent way. A *resolution calculus* gives a formal definition of what it means for a reference to resolve to a declaration. Resolutions are described as paths in the scope graph obeying certain (language-specific) criteria; a given reference may resolve to one or many declarations (or to none). A derived *resolution algorithm* computes the set of declarations to which each reference resolves, and is sound and complete with respect to the calculus.

In this paper, we refine and extend the scope graph framework of [14] to a full framework for static semantic analysis. In essence, this involves uniting a type checker with our existing name resolution machinery. Ideally, we would like to keep these two aspects separated as much as possible for maximum modularity. And indeed, for many language constructs, a simple two-stage approach—name resolution using the scope graph followed by a separate type checking step—would work. But the full story is more complicated, because sometimes name resolution also depends on type resolution. For example, in a language that uses dot notation for object field projection, determining the resolution of `x` in the expression `r.x` requires first determining the object type of `r`, which in turn requires name resolution again. Thus, we require a unified mechanism for expressing and solving arbitrarily interdependent naming and typing resolution problems.

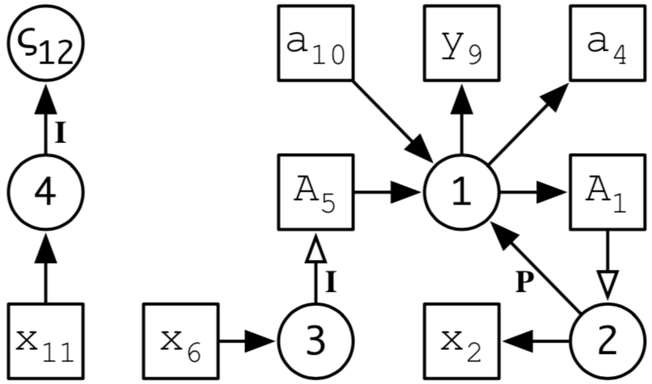
To address this challenge, we base our framework on a language of *constraints*. Term equality constraints are a standard choice for

PEPM'16, January 18–19, 2016, St. Petersburg, FL, USA
ACM. 978-1-4503-4097-7/16/01...$15.00
http://dx.doi.org/10.1145/2847538.2847543

Type-Dependent Name Resolution for Free

```
record A1 { x2 : Int3 }
def a4 = ( new A5{x6=17} )8
def y9 = ( a10.x11 )12
```

Scope graph constraints



Resolution constraints

$$\mathtt{A}_5^{\mathbf{R}} \mapsto \delta_8 \qquad \mathtt{x}_6^{\mathbf{R}} \mapsto \delta_6$$
$$\mathtt{a}_{10}^{\mathbf{R}} \mapsto \delta_{10} \qquad \mathtt{x}_{11}^{\mathbf{R}} \mapsto \delta_{11}$$
$$!\overline{\mathcal{D}}(1) \qquad !\overline{\mathcal{D}}(2)$$
$$\overline{\mathcal{V}}(3) \approx \overline{\mathcal{R}}(3) \qquad \delta_{12} \rightsquigarrow \varsigma_{12}$$

Typing constraints

$$\mathtt{x}_2^{\mathbf{D}} : \tau_3 \qquad \tau_3 \equiv \mathit{Int}$$
$$\mathtt{a}_4^{\mathbf{D}} : \tau_8 \qquad \tau_8 \equiv \mathit{Rec}(\delta_8)$$
$$\mathtt{y}_9^{\mathbf{D}} : \tau_{12} \qquad \delta_6 : \tau_6$$
$$\delta_{10} : \tau_{10} \qquad \delta_{11} : \tau_{12}$$
$$\tau_7 \equiv \tau_6 \qquad \tau_7 \equiv \mathit{Int}$$
$$\tau_{10} \equiv \mathit{Rec}(\delta_{12})$$

Solution

$$\delta_6 = \delta_{11} = \mathtt{x}_2^{\mathbf{D}}$$
$$\delta_8 = \delta_{12} = \mathtt{A}_1^{\mathbf{D}}$$
$$\delta_{10} = \mathtt{a}_4^{\mathbf{D}} \qquad \varsigma_{12} = 2$$
$$\tau_3 = \tau_6 = \tau_7 = \tau_{12} = \mathit{Int}$$
$$\tau_8 = \tau_{10} = \mathit{Rec}(\mathtt{A}_1^{\mathbf{D}})$$

A Constraint Language for Static Semantic Analysis Based on Scope Graphs

Hendrik van Antwerpen
TU Delft, The Netherlands
h.vanantwerpen@tudelft.nl

Pierre Néron
TU Delft, The Netherlands
p.j.m.neron@tudelft.nl

Andrew Tolmach
Portland State University, USA
tolmach@pdx.edu

Eelco Visser
TU Delft, The Netherlands
visser@acm.org

Guido Wachsmuth
TU Delft, The Netherlands
guwac@acm.org

Abstract

In previous work, we introduced *scope graphs* as a formalism for describing program binding structure and performing name resolution in an AST-independent way. In this paper, we show how to use scope graphs to build static semantic analyzers. We use *constraints* extracted from the AST to specify facts about binding, typing, and initialization. We treat name and type resolution as separate building blocks, but our approach can handle language constructs—such as record field access—for which binding and typing are mutually dependent. We also refine and extend our previous scope graph theory to address practical concerns including ambiguity checking and support for a wider range of scope relationships. We describe the details of constraint generation for a model language that illustrates many of the interesting static analysis issues associated with modules and records.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; D.3.2 [*Programming Languages*]: Language classifications; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs; D.3.4 [*Programming Languages*]: Processors; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages; D.2.6 [*Software Engineering*]: Programming Environments

Keywords Language Specification; Name Binding; Types; Domain Specific Languages; Meta-Theory

1. Introduction

Language workbenches [6] are tools that support the implementation of full-fledged programming environments for (domain-specific) programming languages. Ongoing research investigates how to reduce implementation effort by factoring out language-independent implementation concerns and providing high-level meta-languages for the specification of syntactic and semantic aspects of a language [18]. Such meta-languages should (i) have a clear and clean underlying theory; (ii) handle a broad range of common language features; (iii) be declarative, but be realizable by practical algorithms and tools; (iv) be factored into language-specific and language-independent parts, to maximize re-use; and (v) apply to erroneous programs as well as to correct ones.

In recent work we showed how name resolution for lexically-scoped languages can be formalized in a way that meets these criteria [14]. The name binding structure of a program is captured in a *scope graph* which records identifier declarations and references and their scoping relationships, while abstracting away program details. Its basic building blocks are *scopes*, which correspond to sets of program points that behave uniformly with respect to resolution. A scope contains identifier declarations and references, each tagged with its position in the original AST. Scopes can be connected by edges representing lexical nesting or import of named collections of declarations such as modules or records. A scope graph is constructed from the program AST using a language-dependent traversal, but thereafter, it can be processed in a largely language-independent way. A *resolution calculus* gives a formal definition of what it means for a reference to resolve to a declaration. Resolutions are described as paths in the scope graph obeying certain (language-specific) criteria; a given reference may resolve to one or many declarations (or to none). A derived *resolution algorithm* computes the set of declarations to which each reference resolves, and is sound and complete with respect to the calculus.

In this paper, we refine and extend the scope graph framework of [14] to a full framework for static semantic analysis. In essence, this involves uniting a type checker with our existing name resolution machinery. Ideally, we would like to keep these two aspects separated as much as possible for maximum modularity. And indeed, for many language constructs, a simple two-stage approach—name resolution using the scope graph followed by a separate type checking step—would work. But the full story is more complicated, because sometimes name resolution also depends on type resolution. For example, in a language that uses dot notation for object field projection, determining the resolution of `x` in the expression `r.x` requires first determining the object type of `r`, which in turn requires name resolution again. Thus, we require a unified mechanism for expressing and solving arbitrarily interdependent naming and typing resolution problems.

To address this challenge, we base our framework on a language of *constraints*. Term equality constraints are a standard choice for

PEPM'16, January 18–19, 2016, St. Petersburg, FL, USA
ACM. 978-1-4503-4097-7/16/01...$15.00
http://dx.doi.org/10.1145/2847538.2847543

Type Checkers for Free

Problem

- How to represent generic and parameterized types using scope graphs?

Solution

- Represent (module, record, class) types as scopes

Evaluation

- Specifications of STLC-REC, System F, FGJ

Problem

- How to soundly schedule scope graph construction and resolution?

Solution

- Critical edges determine whether there are more potential dependencies

Problem

- How to derive type checkers from declarative type system specifications

Solution

- A ‘logic progr. language’ with scope graph and unification constraints

Scopes as Types

HENDRIK VAN ANTWERPEN, Delft University of Technology, Netherlands
CASPER BACH POULSEN, Delft University of Technology, Netherlands
ARJEN ROUVOET, Delft University of Technology, Netherlands
EELCO VISSER, Delft University of Technology, Netherlands

Scope graphs are a promising generic framework to model the binding structures of programming languages, bridging formalization and implementation, supporting the definition of type checkers and the automation of type safety proofs. However, previous work on scope graphs has been limited to simple, nominal type systems. In this paper, we show that viewing *scopes as types* enables us to model the internal structure of types in a range of non-simple type systems (including structural records and generic classes) using the generic representation of scopes. Further, we show that relations between such types can be expressed in terms of generalized scope graph queries. We extend scope graphs with scoped relations and queries. We introduce Statix, a new domain-specific meta-language for the specification of static semantics, based on scope graphs and constraints. We evaluate the scopes as types approach and the Statix design in case studies of the simply-typed lambda calculus with records, System F, and Featherweight Generic Java.

CCS Concepts: • **Software and its engineering** → **Semantics**; **Domain specific languages**;

Additional Key Words and Phrases: static semantics, type system, type checker, name resolution, scope graphs, domain-specific language

ACM Reference Format:
Hendrik van Antwerpen, Casper Bach Poulsen, Arjen Rouvoet, and Eelco Visser. 2018. Scopes as Types. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 114 (November 2018), 30 pages. https://doi.org/10.1145/3276484

1 INTRODUCTION

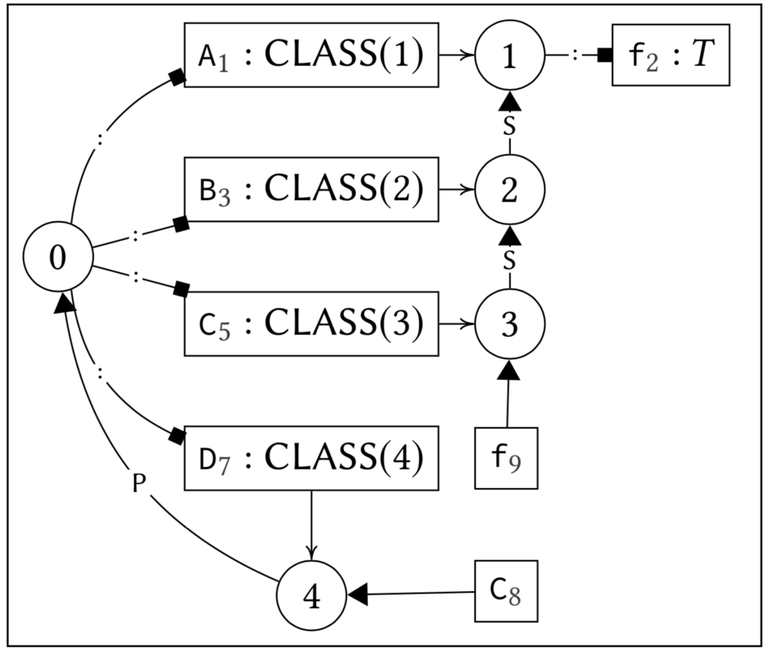
The goal of our work is to support high-level specification of type systems that can be used for multiple purposes, including reasoning (about type safety among other things) and the implementation of type checkers [Visser et al. 2014]. Traditional approaches to type system specification do not reflect the commonality underlying the name binding mechanisms for different languages. Furthermore, operationalizing name binding in a type checker requires carefully staging the traversals of the abstract syntax tree in order to collect information before it is needed. In this paper, we introduce an approach to the declarative specification of type systems that is close in abstraction to traditional type system specifications, but can be directly interpreted as type checking rules. The approach is based on scope graphs for name resolution, and constraints to separate traversal order from solving order.

Authors’ addresses: Hendrik van Antwerpen, Delft University of Technology, Delft, Netherlands, H.vanAntwerpen@tudelft.nl; Casper Bach Poulsen, Delft University of Technology, Delft, Netherlands, C.B.Poulsen@tudelft.nl; Arjen Rouvoet, Delft University of Technology, Delft, Netherlands, A.J.Rouvoet@tudelft.nl; Eelco Visser, Delft University of Technology, Delft, Netherlands, E.Visser@tudelft.nl.

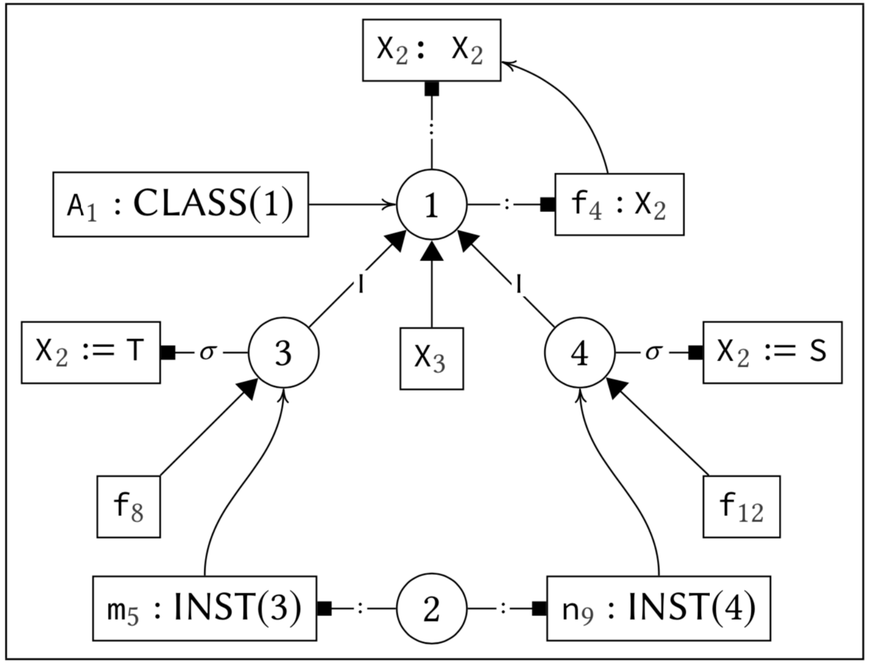
2475-1421/2018/11-ART114
https://doi.org/10.1145/3276484

Type Checkers for Free

```
class A1 { T f2; }
class B3 extends A4 { ... }
class C5 extends B6 { ... }
class D7 { ... new C8().f9 ... }
```




```
class A1<X2> { X3 f4; }
...
m5 = new A6<T>();
m7.f8;
n9 = new A10<S>();
n11.f12;
```



Scopes as Types

HENDRIK VAN ANTWERPEN, Delft University of Technology, Netherlands
CASPER BACH POULSEN, Delft University of Technology, Netherlands
ARJEN ROUVOET, Delft University of Technology, Netherlands
EELCO VISSER, Delft University of Technology, Netherlands

Scope graphs are a promising generic framework to model the binding structures of programming languages, bridging formalization and implementation, supporting the definition of type checkers and the automation of type safety proofs. However, previous work on scope graphs has been limited to simple, nominal type systems. In this paper, we show that viewing *scopes as types* enables us to model the internal structure of types in a range of non-simple type systems (including structural records and generic classes) using the generic representation of scopes. Further, we show that relations between such types can be expressed in terms of generalized scope graph queries. We extend scope graphs with scoped relations and queries. We introduce Statix, a new domain-specific meta-language for the specification of static semantics, based on scope graphs and constraints. We evaluate the scopes as types approach and the Statix design in case studies of the simply-typed lambda calculus with records, System F, and Featherweight Generic Java.

CCS Concepts: • **Software and its engineering → Semantics**; **Domain specific languages**;

Additional Key Words and Phrases: static semantics, type system, type checker, name resolution, scope graphs, domain-specific language

ACM Reference Format:
Hendrik van Antwerpen, Casper Bach Poulsen, Arjen Rouvoet, and Eelco Visser. 2018. Scopes as Types. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 114 (November 2018), 30 pages. https://doi.org/10.1145/3276484

1 INTRODUCTION

The goal of our work is to support high-level specification of type systems that can be used for multiple purposes, including reasoning (about type safety among other things) and the implementation of type checkers [Visser et al. 2014]. Traditional approaches to type system specification do not reflect the commonality underlying the name binding mechanisms for different languages. Furthermore, operationalizing name binding in a type checker requires carefully staging the traversals of the abstract syntax tree in order to collect information before it is needed. In this paper, we introduce an approach to the declarative specification of type systems that is close in abstraction to traditional type system specifications, but can be directly interpreted as type checking rules. The approach is based on scope graphs for name resolution, and constraints to separate traversal order from solving order.

Authors' addresses: Hendrik van Antwerpen, Delft University of Technology, Delft, Netherlands, H.vanAntwerpen@tudelft.nl; Casper Bach Poulsen, Delft University of Technology, Delft, Netherlands, C.B.Poulsen@tudelft.nl; Arjen Rouvoet, Delft University of Technology, Delft, Netherlands, A.J.Rouvoet@tudelft.nl; Eelco Visser, Delft University of Technology, Delft, Netherlands, E.Visser@tudelft.nl.

2475-1421/2018/11-ART114
https://doi.org/10.1145/3276484

Memory Representation for Free

Problem

- Representation of memory in dynamic semantics is ad hoc
- Makes type safety proofs unsystematic

Provide

- Type system using scope graphs for bindings

Get

- A uniform model for memory representation in dynamic semantics

Solution

- Formalization of frames and their correspondence to scopes

Evaluation

- Coq development specifying type system using scope graphs and dynamic semantics using frames for several model languages + type safety proofs

Scopes Describe Frames: A Uniform Model for Memory Layout in Dynamic Semantics*

Casper Bach Poulsen[1], Pierre Néron[2], Andrew Tolmach[†3], and Eelco Visser[4]

1 Delft University of Technology, The Netherlands
c.b.poulsen@tudelft.nl
2 French Network and Information Security Agency (ANSSI), France
pierre.neron@ssi.gouv.fr
3 Portland State University, USA
tolmach@pdx.edu
4 Delft University of Technology, The Netherlands
visser@acm.org

Abstract

Semantic specifications do not make a systematic connection between the names and scopes in the static structure of a program and memory layout, and access during its execution. In this paper, we introduce a systematic approach to the alignment of names in static semantics and memory in dynamic semantics, building on the scope graph framework for name resolution. We develop a uniform memory model consisting of frames that instantiate the scopes in the scope graph of a program. This provides a language-independent correspondence between static scopes and run-time memory layout, and between static resolution paths and run-time memory access paths. The approach scales to a range of binding features, supports straightforward type soundness proofs, and provides the basis for a language-independent specification of sound reachability-based garbage collection.

1998 ACM Subject Classification F.3.1 Specifying and Verifying and Reasoning about Programs

Keywords and phrases Dynamic semantics, scope graphs, memory layout, type soundness, operational semantics

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2016.20

Supplementary Material ECOOP Artifact Evaluation approved artifact available at http://dx.doi.org/10.4230/DARTS.2.1.10

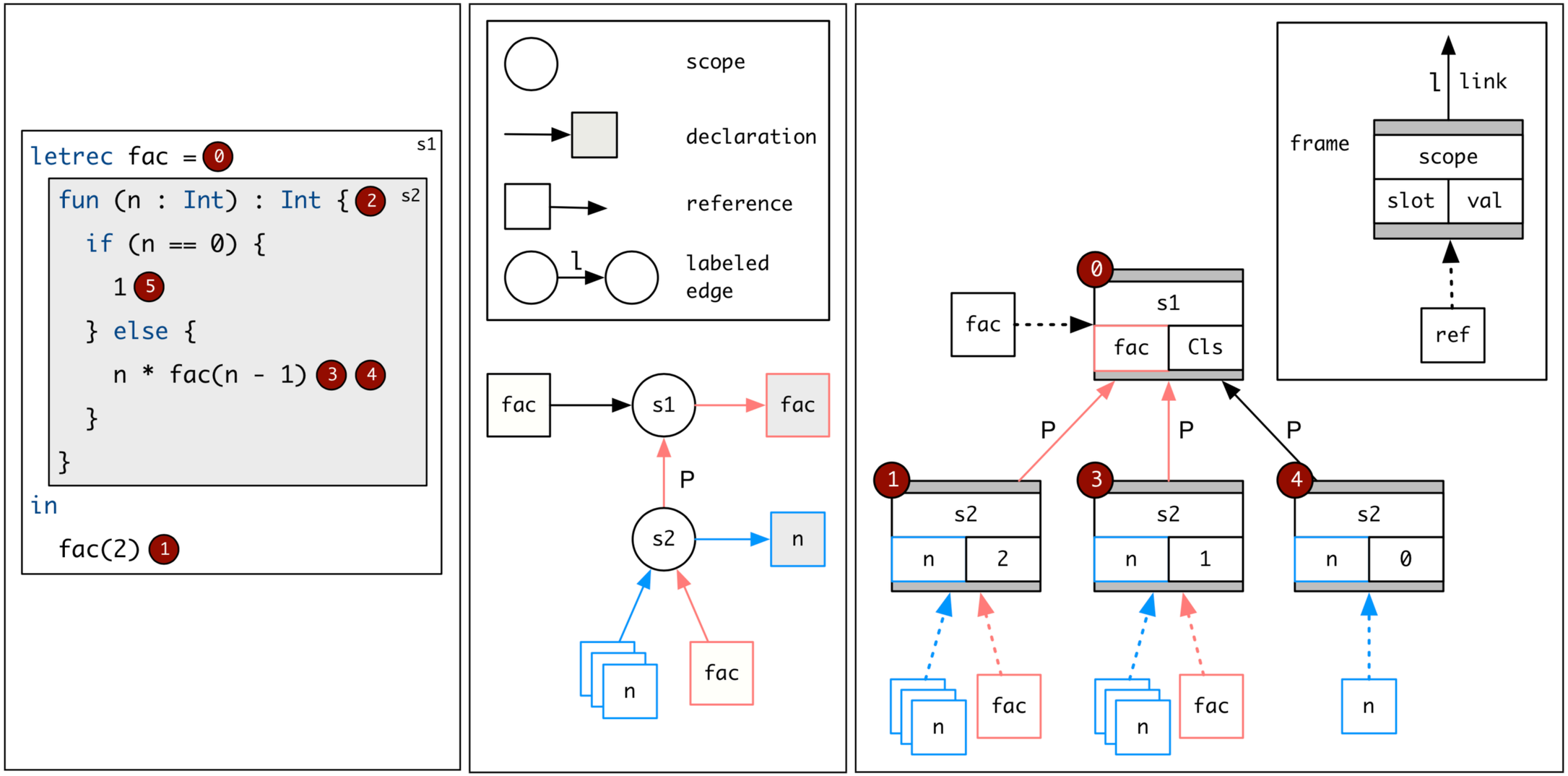
1 Introduction

Name binding and memory management are pervasive concerns in programming language design. There is clearly a connection between the names and scopes in the static structure of a program and patterns of memory allocation, access, and deallocation during its execution. However, existing semantic specifications of programming languages do not treat this connection systematically, and take a wide variety of approaches to handling name binding and

* This research was partially funded by the NWO VICI *Language Designer's Workbench* project (639.023.206).
† Andrew Tolmach was partly supported by a Digiteo Chair at Laboratoire de Recherche en Informatique, Université Paris-Sud.

30th European Conference on Object-Oriented Programming (ECOOP 2016).
Editors: Shriram Krishnamurthi and Benjamin S. Lerner; Article No. 20; pp. 20:1–20:26
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Memory Representation for Free



Type Safety for Free

Problem

- Type safety proofs are tedious
- Dynamic semantics needs to consider ‘bad cases’

Provide

- An intrinsically-typed abstract syntax
- A definitional interpreter defined on that abstract syntax

Get

- A type safety proof

Solution

- Encoding (non-lexical) bindings as part of intrinsically-typed abstract syntax (using scope graphs) in the Agda dependently-typed programming language

Evaluation

- An intrinsically-typed interpreter for Middle-Weight Java

Intrinsically-Typed Definitional Interpreters for Imperative Languages

CASPER BACH POULSEN, Delft University of Technology, The Netherlands
ARJEN ROUVOET, Delft University of Technology, The Netherlands
ANDREW TOLMACH, Portland State University, USA
ROBBERT KREBBERS, Delft University of Technology, The Netherlands
EELCO VISSER, Delft University of Technology, The Netherlands

A definitional interpreter defines the semantics of an object language in terms of the (well-known) semantics of a host language, enabling understanding and validation of the semantics through execution. Combining a definitional interpreter with a separate type system requires a separate type safety proof. An alternative approach, at least for pure object languages, is to use a dependently-typed language to encode the object language type system in the definition of the abstract syntax. Using such intrinsically-typed abstract syntax definitions allows the host language type checker to verify automatically that the interpreter satisfies type safety. Does this approach scale to larger and more realistic object languages, and in particular to languages with mutable state and objects?

In this paper, we describe and demonstrate techniques and libraries in Agda that successfully scale up intrinsically-typed definitional interpreters to handle rich object languages with non-trivial binding structures and mutable state. While the resulting interpreters are certainly more complex than the simply-typed λ-calculus interpreter we start with, we claim that they still meet the goals of being concise, comprehensible, and executable, while guaranteeing type safety for more elaborate object languages. We make the following contributions: (1) A *dependent-passing style* technique for hiding the weakening of indexed values as they propagate through monadic code. (2) An Agda library for programming with *scope graphs* and *frames*, which provides a uniform approach to dealing with name binding in intrinsically-typed interpreters. (3) Case studies of intrinsically-typed definitional interpreters for the simply-typed λ-calculus with references (STLC+Ref) and for a large subset of Middleweight Java (MJ).

CCS Concepts: • **Theory of computation → Program verification**; *Type theory*; • **Software and its engineering → Formal language definitions**;

Additional Key Words and Phrases: definitional interpreters, dependent types, scope graphs, mechanized semantics, Agda, type safety, Java

ACM Reference Format:
Casper Bach Poulsen, Arjen Rouvoet, Andrew Tolmach, Robbert Krebbers, and Eelco Visser. 2018. Intrinsically-Typed Definitional Interpreters for Imperative Languages. *Proc. ACM Program. Lang.* 2, POPL, Article 16 (January 2018), 34 pages. https://doi.org/10.1145/3158104

Authors' addresses: Casper Bach Poulsen, Delft University of Technology, The Netherlands, c.b.poulsen@tudelft.nl; Arjen Rouvoet, Delft University of Technology, The Netherlands, a.j.rouvoet@tudelft.nl; Andrew Tolmach, Portland State University, Oregon, USA, tolmach@pdx.edu; Robbert Krebbers, Delft University of Technology, The Netherlands, r.j.krebbers@tudelft.nl; Eelco Visser, Delft University of Technology, The Netherlands, e.visser@tudelft.nl.

2475-1421/2018/1-ART16
https://doi.org/10.1145/3158104

Type Safety for Free

$$\mathsf{eval} \;:\; \mathbb{N} \to \forall\,\{s\ t\ \Sigma\} \to \mathsf{Expr}\ s\ t \to \mathsf{M}\ s\ (\mathsf{Val}\ t)\ \Sigma$$

```
data Expr (s : Scope) : Ty → Set where
  var     : ∀ {t} → (s ↦ v^t t) → Expr s t
  new     : ∀ {s^r s'} → s ↦ c^t s^r s' → Expr s (ref s')
  get     : ∀ {s' t} → Expr s (ref s') → (s' ↦ v^t t) → Expr s t
  call    : ∀ {s' ts t} → Expr s (ref s') → (s' ↦ (m^t ts t)) →
            All (Expr s) ts → Expr s t
  upcast  : ∀ {s' s''} → s' <: s'' → Expr s (ref s') → Expr s (ref s'')
```

```
eval (suc k) (new x)          = getv x >>= λ{ (c^t class f) →
                                usingFrame f (init-obj k class) >>= λ{ f' →
                                return (ref [] f') }}
eval (suc k) (get e p)        = eval k e >>= λ{ null → raise ; (ref p' f) →
                                usingFrame f (getv (prepend p' p)) >>= λ{ (v^t v) →
                                return v }}
eval (suc k) (call e p args)  = eval k e >>= λ { null → raise ; (ref p' f) →
                                usingFrame f (getv (prepend p' p)) >>= λ{ (m^t f' (meth s b)) →
                                (eval-args k args ^ f') >>= λ{ (slots , f') →
                                init s slots (f' :: []) >>= λ f' →
                                usingFrame f' (eval-body k b) }}}
eval (suc k) (upcast p e)     = eval k e >>= λ v → return (upcastRef p v)
```

More Programming Stuff for Free (Work in Progress)

Type checkers for languages with gradual / substructural / … type systems for free

- Question: How to turn declarative specifications for these type systems into type checkers?

Type safety for languages with gradual / substructural / … type systems for free

- Question: can we extend the intrinsically-typed definitional interpreter approach to more expressive type systems?

Program refactorings for free

- Idea: refactorings can be specified by constraints

Incremental modular type checkers for free

- Idea: module dependencies determined by scope graph

Control for free

- Question: what is the analog of scopes-as-frames for control?

One topic, a range of PL methods

Domain

- How does programming programming environments work?

Design

- Exploring language designs for better abstractions for name binding and resolution

Specification

- Formal semantics of name resolution, correctness of algorithms wrt semantics
- Specification of dynamic semantics, (automatic) type soundness proofs

Implementation

- Language implementation, constraint solvers, IDE integration

Evaluation

- Benchmarking performance, evaluating coverage, informally testing usability

PL research is about getting programming stuff for free

Your Turn

Go out into the world (but first SPLASH) and test this theory

Confirm

- Can you identify programming languages research that fits in this framework?
- What is the problem they are trying to solve?
- What is the solution providing for free?
- etc.

Refute

- Can you identify programming languages research that ***does not fit*** in this framework?
- What kind of research is it?
- Why does it not fit?

Ask your colleagues / advisors: what is PL research?