

Lecture 7: Type Checking

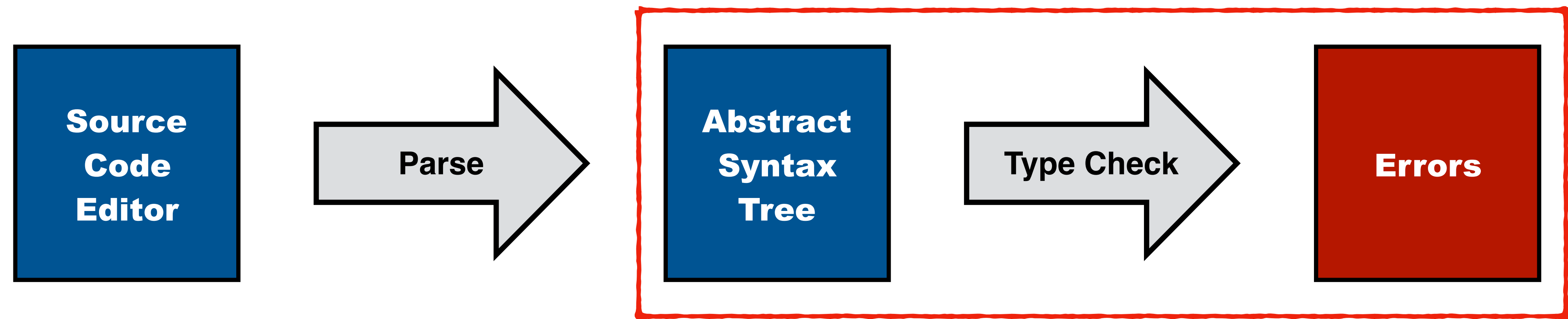
Hendrik van Antwerpen

CS4200 Compiler Construction

TU Delft

October 2018

This Lecture



Check that names are used correctly and that expressions are well-typed

Reading Material

The following papers add background, conceptual exposition, and examples to the material from the slides. Some notation and technical details have been changed; check the documentation.

Type checkers are algorithms that check names and types in programs.

This paper introduces the NaBL Name Binding Language, which supports the declarative definition of the name binding and scope rules of programming languages through the definition of scopes, declarations, references, and imports associated.

Background

SLE 2012

http://dx.doi.org/10.1007/978-3-642-36089-3_18

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 Spoofox 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.
© Springer-Verlag Berlin Heidelberg 2013

This paper introduces scope graphs as a language-independent representation for the binding information in programs.

Best EAPLS paper at ETAPS 2015

ESOP 2015

http://dx.doi.org/10.1007/978-3-662-46669-8_9

A Theory of Name Resolution

Pierre Neron¹, Andrew Tolmach², Eelco Visser¹, and Guido Wachsmuth¹

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

Separating type checking into constraint generation and constraint solving provides more declarative definition of type checkers. This paper introduces a constraint language integrating name resolution into constraint resolution through scope graph constraints.

This is the basis for the design of the NaBL2 static semantics specification language.

PEPM 2016

<https://doi.org/10.1145/2847538.2847543>

A Constraint Language for Static Semantic Analysis based on Scope Graphs

Hendrik van Antwerpen
Delft University of Technology
h.vanantwerpen@student.tudelft.nl

Pierre Néron
Delft University of Technology
p.j.m.neron@tudelft.nl

Andrew Tolmach
Portland State University
tolmach@pdx.edu

Eelco Visser
Delft University of Technology
visser@acm.org

Guido Wachsmuth
Delft University of Technology
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.

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 show 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, none, or many declarations. 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 construct 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, our framework requires 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 describing type inference problems while abstracting away from the details of an AST in a particular language. Adopting constraints to describe both typing and scoping requirements has the advantage of uniform notation, and, more importantly, provides a clean way to combine naming and typing problems. In particular, we extend our previous work to support *incomplete* scope graphs, which correspond to constraint sets with (as yet) unresolved variables.

Our new framework continues to satisfy the criteria outlined above. (i) The resolution calculus and standard term equality constraint theory provide a solid language-independent theory for name and type resolution. (ii) Our framework supports type checking and inference for statically typed, monomorphic languages with user-defined types, and can also express uniqueness and completeness requirements on declarations and initializers. The framework inherits from scope graphs the ability to model a broad range of binding patterns, including many variants of lexical scoping, records, and modules. (iii) The constraint language has a declarative semantics given by a constraint satisfaction relation, which employs the resolution calculus to define the semantics of name resolution relative to a scope graph. We define a constraint resolution algorithm based on our previous name resolution algorithm,

Documentation for NaBL2 at the metaborg.org website.

<http://www.metaborg.org/en/latest/source/langdev/meta/lang/nabl2/index.html>

 Spoofax
latest

Search docs

The Spoofax Language Workbench
 Examples
 Publications

TUTORIALS

Installing Spoofax
 Creating a Language Project
 Using the API
 Getting Support

REFERENCE MANUAL

Language Definition with Spoofax
 Abstract Syntax with ATerms
 Syntax Definition with SDF3
Static Semantics with NaBL2
 1. Introduction
 2. Language Reference
 3. Configuration
 4. Examples
 5. Bibliography
 6. NaBL/TS (Deprecated)
 Transformation with Stratego
 Dynamic Semantics with DynSem
 Editor Services with ESV
 Language Testing with SPT
 Building Languages
 Programmatic API
 Developing Spoofax

[ing/nabl2/index.html](#)

Development Release
 Release Archive
 Migration Guides

Docs » Static Semantics Definition with NaBL2

Edit on GitHub

Static Semantics Definition with NaBL2

Programs that are syntactically well-formed are not necessarily valid programs. Programming languages typically impose additional *context-sensitive* requirements on programs that cannot be captured in a syntax definition. Languages use names to identify reusable units that can be invoked at multiple parts in a program. In addition, statically typed languages require that expressions are consistently typed. The NaBL2 ‘Name Binding Language’ supports the specification of name binding and type checking rules of a language. NaBL2 uses a constraint-based approach, and uses scope graphs for name resolution.

Table of Contents

- 1. Introduction
 - 1.1. Name Resolution with Scope Graphs
- 2. Language Reference
 - 2.1. Lexical matters
 - 2.2. Modules
 - 2.3. Signatures
 - 2.4. Rules
 - 2.5. Constraints
- 3. Configuration
 - 3.1. Prepare your project
 - 3.2. Runtime settings
 - 3.3. Customize analysis
 - 3.4. Inspecting analysis results
- 4. Examples
- 5. Bibliography
- 6. NaBL/TS (Deprecated)
 - 6.1. Namespaces
 - 6.2. Name Binding Rules
 - 6.3. Interaction with Type System

Note

The predecessor of NaBL2, the NaBL/TS name binding and type analysis meta-language is deprecated.

This paper describes the next generation of the approach.

Addresses (previously) open issues in expressiveness of scope graphs for type systems:

- Structural types
- Generic types

Addresses open issue with staging of information in type systems.

Introduces Statix DSL for definition of type systems.

Prototype of Statix is available in Spoofax HEAD, but not ready for use in project yet.

The future

OOPSLA 2018

To appear

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.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

© 2018 Copyright held by the owner/author(s).
2475-1421/2018/11-ART114
<https://doi.org/10.1145/3276484>

Types

Why types?

Why types?

- Last week: "guarantee absence of run-time type errors"

What is a type system?

- A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute. [Pierce2002]

Discuss using a series of untyped examples

- Do you consider the example correct or not, and why?
 - ▶ That is, do you think it should type-check?
- If incorrect: what types will disallow this program?
- If correct: what types will allow this program?

Preliminaries

```
class A {  
  B b;  
  int m(int i) {  
    return i + b.f  
  }  
}  
  
class B {  
  int f;  
}
```

How do types show up in programs?

- Type literals describe types
- Type definitions introduce new (named) types
- Type references refer to named types
- Declared variables have types ($x : T$)
- Expressions have types ($e : T$)
 - Including all sub-expressions

Types Example

4 / "four"

4 : number
"four" : string
/ : number * number → number

simple types

typing prevents undefined runtime behavior

Types Example

```
7 + (if (true) { 5 } else { "four" })
```

7 : number

5 : number

"four" : boolean

if : ?

no simple type

- typing (over)approximates runtime behavior
- programs without runtime errors can be rejected

Types Example

```
function id(x) { return x; }  
id(4); id(true);
```

```
4      : number  
true   : boolean  
id     :  $\forall T. T \rightarrow T$ 
```

polymorphic type

- richer types approximate behavior better
- depends on runtime representation of values

Types Example

```
if (a < 5) { 5 } else { "four" }
```

```
5      : number  
"four" : string  
if     : number|string
```

union type

- richer types approximate behavior better
- depends on runtime representation of values

Types Example

```
float distance = 12.0, time = 4.0  
float velocity = time / distance
```

```
distance : float<m>  
time      : float<s>  
velocity  : number<m/s>
```

unit-of-measure type

- no runtime problems, but not correct
- types can enforce other correctness properties

What kind of types?

- Simple `int, float → float, bool`
- Named `class A, newtype Id`
- Polymorphic `List<X>, ∀a. a → a`
- Union/sum (one of) `string | string[]`
- Unit-of-measure `float<m>, float<m/s>`
- Structural `{ x: number, y: number }`
- Intersection (all of) `Comparable & Serializable`
- Recursive `μT. int | T * T`
- Ownership `&mut data`
- Dependent – values in types `Vector 3`
- ... many more ...

Why types?

Why types?

- Statically prove the absence of certain (wrong) runtime behavior
 - ▶ “Well-typed programs cannot go wrong.” [Reynolds1985]
 - ▶ Also logical properties beyond runtime problems

What are types?

- Static classification of expressions by approximating the runtime values they may produce
- Richer types approximate runtime behavior better
- Richer types may encode correctness properties beyond runtime crashes

What is the difference between typing and testing?

- Typing is an over-approximation of runtime behavior (proof of absence)
- Testing is an under-approximation of runtime behavior (proof of presence)

Types and language design

Types influence language design

- Types abstract over implementation
 - ▶ Any value with the correct type is accepted
- Types enable separate or incremental compilation
 - ▶ As long as the public interface is implemented, dependent modules do not change

Can we have our cake and eat it too?

- Ever more precise types lead to ever more correct programs
- What would be the most precise type you can give?
 - ▶ The exact set of values computed for a given input?
- Expressive typing problems become hard to compute
- Many are undecidable, if they imply solving the halting problem
- Designing type systems always involves trade-offs

Relations between Types

Comparing Types

```
interface Point2D { x: number, y: number }  
interface Vector2D { x: number, y: number }  
var p1: Point2D = { x: 5, y: -11 }  
var p2: Vector2D = p1
```

Is this program correct?

- No, if types are compared by name
- Yes, if types are compared based on structure

Comparing Types

```
interface Point2D { x: number, y: number }  
interface Point3D { x: number, y: number, z: number }  
var p1: Point3D = { x: 5, y: -11, z: 0 }  
var p2: Point2D = p1
```

Is this program correct?

- No, if equal types are required
- Yes, if structural subtypes are allowed
- When is T a subtype of U?
 - ▶ When a value of type T can be used when a value of U is expected
- What about nominal subtypes?
 - ▶ `interface Point3D extends Point2D`

Combination example: generics and subtyping

```
class A {}  
class B extends A {}  
  
B[] bs = new B[1];  
A[] as = bs;  
as[0] = new A();  
B b = bs[0];
```

*subtyping on arrays &
mutable updates is unsound*

- unsound = under-approximation of runtime behavior
- feature combinations are not trivial

Comparing Types

```
int i = 12  
float f = i
```

Is this program correct?

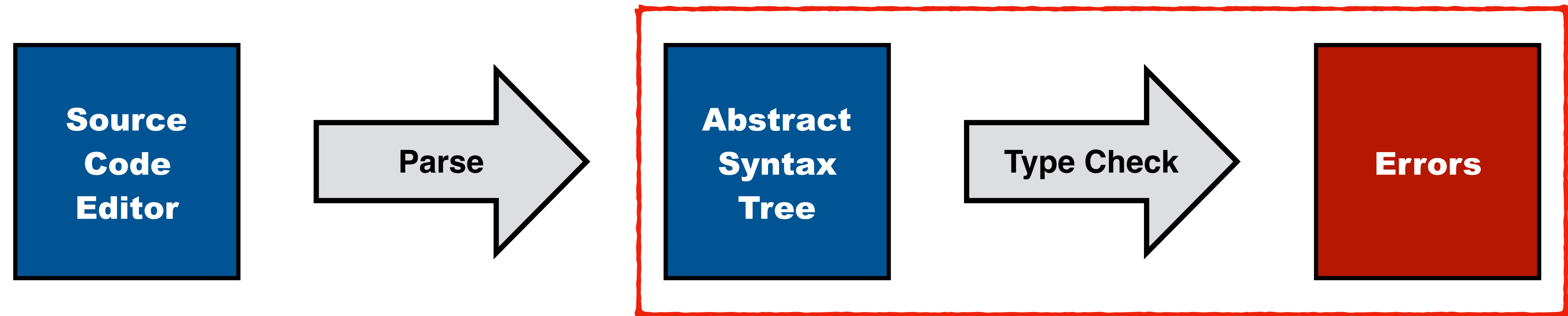
- No, floats and integers have different runtime representations
- Yes, possible by coercion
 - ▶ Coercion requires insertion of code to convert between representations
- How is this different than subtyping?
 - ▶ Subtyping says that the use of the unchanged value is safe

Type Relations

What kind of relations between types?

- Equality $T = T$ – syntactic or structural
- Subtyping $T < : T$ – nominal or structural
- Coercion – requires code insertion

Type Checking



Check that names are used correctly and that expressions are well-typed

Type system of a language

What is a type system?

- A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute. [Pierce2002]

Type system specification

- (Formal) description of the typing rules
- For human comprehension
- To prove properties

Type checking algorithm

- Executable version of the specification
- Often contains details that are omitted from the specification

Developing both is a lot of work, often only an algorithm

How to check types?

What should a type checker do?

- Check that a program is well-typed!
- Resolve names, and check or compute types
- Report useful error messages
- Provide a representation of name and type information
 - ▶ Type annotated AST

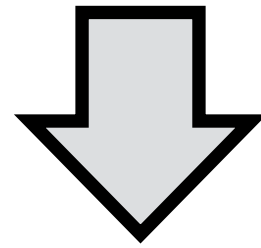
This information is used for

- Next compiler steps (optimization, code generation, ...)
- IDE (error reporting, code completion, refactoring, ...)
- Other tools (API documentation, ...)

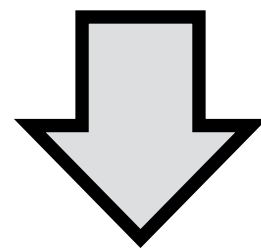
How are type checkers implemented?

Computing Type of Expression (recap)

```
function (a : int) = a + 1
```



```
Fun("i", INT(),  
    Plus(Var("i"), Int(1)))
```



```
FUN(INT(), INT())
```

rules

type-check(lenv):

Fun(x, t, e) -> FUN(t, t')

where <type-check(l[(x, t) | lenv])> e => t'

type-check(lenv):

Var(x) -> t

where <lookup> (x, env) => t

type-check(lenv):

Int(_) -> INT()

type-check(lenv):

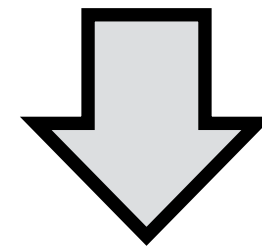
Plus(e1, e2) -> INT()

where <type-check(lenv)> e1 => INT()

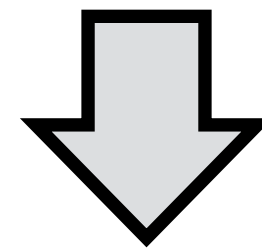
where <type-check(lenv)> e2 => INT()

Inferring the Type of a Parameter

```
function (a       ) = a + 1
```



```
Fun("i",         
    Plus(Var("i"), Int(1)))
```



```
FUN(INT(), INT())
```

rules

Unknown t !

type-check(l env):

Fun(x , e) \rightarrow FUN(t , t')
where \langle type-check($l[(x, t) \mid env]$) $\rangle e \Rightarrow t'$

type-check(l env):

Var(x) $\rightarrow t$
where \langle lookup $\rangle (x, env) \Rightarrow t$

type-check(l env):

Int($_$) \rightarrow INT()

type-check(l env):

Plus($e1$, $e2$) \rightarrow INT()
where \langle type-check(l env) $\rangle e1 \Rightarrow$ INT()
where \langle type-check(l env) $\rangle e2 \Rightarrow$ INT()

- What are the consequences for our algorithm?
- Our types are not known from the start, but learned gradually
- A simple down-up traversal is insufficient

Checking classes

```
class A {  
    B m() {  
        return new C();  
    }  
}  
  
class B {  
    int i;  
}  
  
class C extends B {  
    int m(A a) {  
        return a.m().i;  
    }  
}
```

How can we type check this program?

- Is there a possible down-up strategy here?
- Why are the type annotations not enough?
- What strategy could be used?

Two-pass approach

- The first pass builds a class table
- The second pass checks expressions using the class table
- Does this still work if we introduce nested classes?

How to check types?

What are challenges when implementing a type checker?

- Collecting necessary binding information before using it
- Gradually learning type information

What are the consequences of these challenges?

- The order of computation needs to be more flexible than the AST traversal

Type Checking with Constraints

Constraint-based type checking

Two phase approach

- First record constraints (type equality, name resolution)
- Then solve constraints

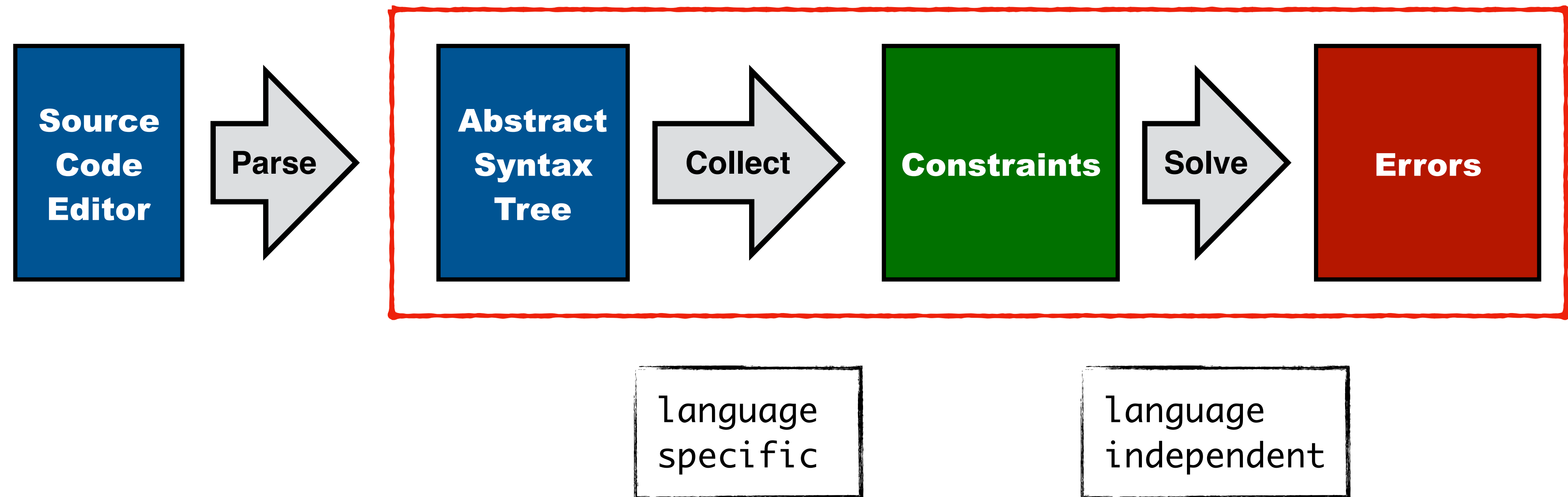
Separation of concern

- Language specific specification in terms of constraints
- Language independent algorithm to solve constraints
- Write specification, get an executable checker

Advantages

- Order of solving independent of order of collection
- Natural support for inference
- Many constraint-based formulations of typing features exist
- Constraint variables act as interface between different kinds of constraints

Combining different kinds of constraints is still non-trivial!



Type checking proceeds in two steps

Use Variables and Constraints

rules

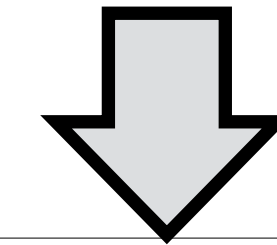
```
type-con(env):  
  Fun(x, e) -> (FUN(t, t'), C)  
  where !VAR(<fresh>) => t;  
  <type-con([x, t] env)> e => (t', C)
```

```
type-con(env):  
  Var(x) -> (t, [])  
  where <lookup> (x, env) => t
```

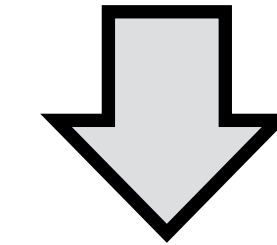
```
type-con(env):  
  Int(_) -> (INT(), [])
```

```
type-con(env):  
  Plus(e1, e2) -> (INT(), [ C1*, C2*,  
                               Eq(t1, INT()),  
                               Eq(t2, INT()) ])  
  where <type-con(env)> e1 => (t1, C2*)  
  where <type-con(env)> e2 => (t2, C1*)
```

function (a) = a + 1



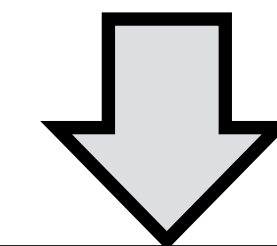
Fun("i", Plus(Var("i"), Int(1)))



FUN(VAR("a") INT())

+

Eq(VAR("a"), INT())
Eq(INT(), INT())



VAR("a") => INT()

Constraint solving order != constraint generation order

Conclusion

Summary

Types

- Static approximation of runtime values computed by expressions
- Prove correctness properties, such as the absence of runtime errors

Type Checking

- Check if programs are well-typed, according to the type system rules
- Challenges around computation order and partial information

Constraint-based type checking

- Language-specific specification
- Language-independent solver
- Resolution order is independent of the AST traversal order

Except where otherwise noted, this work is licensed under

