Declarative Name Binding for Type System Specifications

van Antwerpen, H.

DOI
10.4233/uuid:4bf44aa1-779c-4a96-8c55-5e1b54e16119

Publication date
2024

Document Version
Final published version

Citation (APA)
van Antwerpen, H. (2024). *Declarative Name Binding for Type System Specifications*. [Dissertation (TU Delft), Delft University of Technology]. https://doi.org/10.4233/uuid:4bf44aa1-779c-4a96-8c55-5e1b54e16119

Declarative Name Binding for Type System Specifications

Hendrik van Antwerpen

Declarative Name Binding for Type System Specifications

Declarative Name Binding for Type System Specifications

Dissertation

for the purpose of obtaining the degree of doctor
at Delft University of Technology
by the authority of the Rector Magnificus prof. dr. ir. T.H.J.J. van der Hagen;
Chair of the Board of Doctorates
to be defended publicly on
Wednesday, 15 January 2025 at 17:30 o'clock

by

Hendrik van Antwerpen
Master of Science in Computer Science,
Delft University of Technology, The Netherlands
born in Rotterdam, The Netherlands

This dissertation has been approved by the promotors.

Composition of the doctoral committee:

Rector Magnificus	chairperson
Prof. dr. A. van Deursen	Delft University of Technology, promotor
Dr. J.G.H. Cockx	Delft University of Technology, copromotor
Independent members:	
Prof. dr. A.P. Tolmach	Portland State University, USA
Prof. dr. R.B. Findler	Northwestern University, USA
Prof. dr. G. Hedin	Lund University, Sweden
Dr. W.S. Swierstra	Utrecht University
Prof. dr. M.T.J. Spaan	Delft University of Technology
Prof. dr. A.E. Zaidman	Delft University of Technology (reserve member)

Prof. dr. Eelco Visser (Delft University of Technology) was the original promotor and supervisor of this research until his untimely passing on 5 April 2022.

The work in this dissertation has been carried out at the Delft University of Technology under the auspices of the research school IPA (Institute for Programming research and Algorithmics). This research was funded by the NWO VICI Language Designer's Workbench project (639.023.206).

Typesetting in LaTeX using Semplicità, Palatino, Source Sans 3, and Source Code Pro.
Cover design: © Evelien Jagtman – https://evelienjagtman.com.
Printing and binding: Gildeprint – https://www.gildeprint.nl.

ISBN 978-94-6384-704-9
IPA Dissertation Series 2025-01

An electronic version of this dissertation is available at https://repository.tudelft.nl.

To Nina, my most supportive reviewer 2

What does it mean

Czesław Miłosz

It does not know it glitters
It does not know it flies
It does not know it is this not that.

And, more and more often, agape,
With my Gauloise dying out,
Over a glass of red wine,
I muse on the meaning of being this not that.

Just as long ago, when I was twenty,
But then there was a hope I would be everything,
Perhaps even a butterfly or a thrush, by magic.
Now I see dusty district roads
And a town where the postmaster gets drunk every day
Melancholy with remaining identical to himself.

If only the stars contained me.
If only everything kept happening in such a way
That the so-called world opposed the so-called flesh.
Were I at least not contradictory. Alas.

Contents

Summary

Name binding is an integral part of the static semantics of programming languages. Modern languages commonly feature name binding constructs, such as packages, modules, and user-defined types that are essential to develop and maintain large programs. Static reasoning about name binding is key to support many services offered by modern programming environments, from type checking, to interactive code navigation and automatic refactoring. Formal specification of a programming language is important to understand and reason about the language, as well as to implement it.

Expressive name binding features pose challenges for both high-level specifications and implementations. The first challenge is finding high-level abstractions to describe expressive name binding. The second challenge is correctly implementing high-level specifications that were not written with the goal of implementation in mind. Due to these challenges, specifications are often restricted to a core language that lacks many of the surface language's features. The surface language is only defined by the details of the lower-level implementation. Those implementations use specialized data structures and algorithms to support the full language, which limits code reuse, increases development effort, and makes it more difficult to ensure correctness.

Meta-languages aim to address these challenges and bridge the gap between high-level specification and implementation by providing reusable abstractions for common aspects of programming languages, as well as reusable implementations for specifications in the meta-language. Reusable implementations greatly reduce development effort, and their correctness has to be shown only once instead of for each individual language implementation.

This dissertation proposes a novel meta-language, Statix, for the specification of static semantics. It is based on scope graphs, a general model for name binding in programming languages (Néron et al., 2015). Statix supports the direct modeling of surface language name binding features, stays close to a familiar inference-style of specification, and allows automatically deriving implementations for compilers and editor services. This dissertation makes the following three contributions.

First, we present the design of the meta-language Statix. Statix is a logic language extended with primitives to construct and query scope graphs. Specifications are written as logical predicates that abstract over evaluation order. The design is accompanied by a declarative semantics that gives a precise description of the meaning of specifications written in the meta-language.

Second, we present an operational semantics for Statix that executes specifications in the meta-language as type checkers. We prove that the operational semantics is correct with respect to the declarative semantics. We observe that the operational semantics is incomplete, but argue based on our experience with case studies that this is rarely a problem in practice. We present a framework to implement implicitly parallelized scope-graph-based type checkers. We apply this to our Statix implementation, resulting in substantially improved runtime performance. Additionally, we propose an approach towards implementing language-parametric semantic editor services based on meta-language specifications.

Third, we implement these operational semantics and present case studies of several core languages from the literature as well as of a large subset of Java. The case studies consist of specifications and executable test suites, which allows us to evaluate both the expressiveness and the practical usability of the meta-language.

We evaluate our work by assessing whether Statix (i) has a clear and clean underlying theory (*principled*); (ii) can handle a broad range of common language features (*expressive*); (iii) is *declarative*, but realizable by practical algorithms and tools (*executable*); (iv) is factored into language-specific and language-independent parts, to maximize reuse (*reusable*); and (v) can be applied to erroneous programs as well as to correct ones (*resilient*). We conclude that our approach is sufficient to express and interpret common name binding and type system features. The approach scales to the full surface syntax of real-world programming languages. As such the meta-language fulfills criteria (i) to (iv) well, although we do identify potential improvements. Criterion (v) is only minimally met and remains an important open challenge.

Our work shows that high-level specification of surface language name binding features using a meta-language approach is feasible and useful. It allows easier specification of complete languages, and supports reusable practical implementations for those languages. The following three topics are important areas of future research. The first is to develop abstractions for language features that are currently hard or cumbersome to specify in Statix. The second is to improve the integration of Statix in development environments. This requires better handling of erroneous programs and improved error reporting. It also requires developing program representations that make static program information accessible for program transformation and compilation. The third is to develop approaches for code completion, program repair, and program generation based on Statix specifications. Further research on these topics would make this meta-language approach even more widely applicable.

Samenvatting

Naambinden is een integraal onderdeel van de statische semantiek van programmeertalen. Moderne talen bevatten vaak naambindingsconstructies, zoals pakketten, modules en door de gebruiker gedefinieerde types, die essentieel zijn voor het ontwikkelen en onderhouden van grote programma's. Statisch redeneren over naambinden is cruciaal voor het ondersteunen van vele diensten die moderne programmeeromgevingen bieden, zoals typechecken, interactieve codenavigatie en automatisch refactoren. Formele specificatie van een programmeertaal is belangrijk om de taal te begrijpen, over de taal te redeneren en om deze te implementeren.

Expressieve naambindingsconstructies vormen uitdagingen zowel voor hoogniveau-specificaties als voor implementaties. De eerste uitdaging is de juiste abstracties te vinden om expressieve naambinding op hoog niveau uit te drukken. De tweede uitdaging is het correct implementeren van hoogniveau-specificaties wanneer deze specificaties oorspronkelijk niet voor dit doel geschreven zijn. Vanwege deze uitdagingen worden specificaties vaak beperkt tot een kerntaal die veel kenmerken van de oppervlaktetaal achterwege laat. De oppervlaktetaal is dan alleen gedefinieerd door de details van de lagerniveau-implementatie. Die implementaties gebruiken gespecialiseerde datastructuren en algoritmen om de volledige taal te ondersteunen, wat de herbruikbaarheid van code beperkt, de ontwikkelinspanning vergroot en het moeilijker maakt om de correctheid te garanderen.

Meta-talen beogen deze uitdagingen op te lossen en het gat tussen hoogniveau-specificatie en implementatie te dichten door middel van herbruikbare abstracties voor gemeenschappelijke aspecten van programmeertalen, en herbruikbare implementaties voor specificaties in de meta-taal. Herbruikbare implementaties verminderen de ontwikkelinspanning aanzienlijk, en hun correctheid hoeft slechts éénmaal te worden aangetoond en niet meer voor elke individuele taalimplementatie.

Dit proefschrift stelt een nieuwe meta-taal, Statix, ten behoeve van de specificatie van statische semantiek voor. De meta-taal is gebaseerd op scopegrafen, een algemeen model voor naambinden in programmeertalen (Néron e.a., 2015). Statix ondersteunt het direct modelleren van naambindingskenmerken van oppervlaktetalen, blijft dicht bij een vertrouwde inferentie-stijl van specificeren, en maakt het mogelijk om implementaties voor compilers en ondersteuning voor ontwikkelomgevingen automatisch af te leiden. Dit proefschrift levert de volgende drie bijdragen.

Ten eerste presenteren we het ontwerp van de meta-taal Statix. Statix is een logische taal uitgebreid met primitieven om scopegrafen te construeren en te queryen. Specificaties worden geschreven als logische predicaten die abstraheren over de volgorde van evaluatie. Samen met het ontwerp geven we een declaratieve semantiek die de betekenis van specificaties in de meta-taal precies beschrijft.

Ten tweede presenteren we een operationele semantiek voor Statix die specificaties in de meta-taal uitvoert als typecheckers. We bewijzen dat deze operationele semantiek correct is ten opzichte van de declaratieve semantiek. We merken op dat de operationele semantiek onvolledig is, maar stellen op basis van onze ervaringen met casestudies vast dat dit in de praktijk zelden een probleem is. We presenteren een framework voor het implementeren van impliciet geparalleliseerde typecheckers op basis van scopegrafen. Dit passen we toe op onze Statix-implementatie, wat leidt tot aanzienlijk verbeterde executietijden. Daarnaast stellen we een aanpak voor om taalparametrische semantische ondersteuning voor ontwikkelomgevingen te implementeren op basis van meta-taalspecificaties.

Ten derde implementeren we deze operationele semantiek en presenteren we casestudies voor verschillende kerntalen uit de literatuur en voor een groot deel van Java. De casestudies bestaan uit specificaties en uitvoerbare testcases, waarmee we zowel de expressiviteit als de praktische bruikbaarheid van de meta-taal evalueren.

We evalueren ons werk door te beoordelen of Statix (i) een duidelijke en zuivere onderliggende theorie heeft (*principieel*); (ii) een breed scala aan veelvoorkomende taalkenmerken aankan (*expressief*); (iii) *declaratief* is, maar realiseerbaar met praktische algoritmen en tools (*uitvoerbaar*); (iv) is opgedeeld in taalspecifieke en taalonafhankelijke delen, om hergebruik te maximaliseren (*herbruikbaar*); en (v) kan worden toegepast op zowel foutieve als correcte programma's (*robuust*). We concluderen dat onze aanpak voldoende is om veelvoorkomende naambindings- en typesysteemkenmerken uit te drukken en te interpreteren. De aanpak schaalt naar de volledige oppervlaktesyntax van reële programmeertalen. Als zodanig voldoet de meta-taal goed aan criteria (i) tot (iv), hoewel er mogelijke verbeteringen zijn. Aan criterium (v) wordt slechts minimaal voldaan en dit blijft een belangrijke open uitdaging.

Ons werk laat zien dat hoogniveau-specificatie van naambindingsfuncties van oppervlaktetalen met behulp van een meta-taalbenadering haalbaar en nuttig is. De meta-taal maakt het gemakkelijker om complete talen te specificeren en ondersteunt herbruikbare praktische implementaties voor die talen. De volgende drie onderwerpen zijn belangrijke gebieden voor toekomstig onderzoek. De eerste is het ontwikkelen van abstracties voor taalkenmerken die momenteel moeilijk of alleen omslachtig te specificeren zijn in Statix. De tweede is het verbeteren van de integratie van Statix in ontwikkelomgevingen. Dit vereist een betere aanpak voor foutieve programma's en verbeterde foutmeldingen. Hiervoor zijn programmarepresentaties nodig die statische programma-informatie toegankelijk maken voor programmatransformatie en -compilatie. De derde is het ontwikkelen van benaderingen voor codesuggesties, alsook programmareparatie en -generatie op basis van Statix-specificaties. Verder onderzoek kan deze meta-taalaanpak nog breder toepasbaar maken.

Acknowledgments

Quand on a goûté à la langue, on s'ennuie assez vite avec toute autre forme de langage.

Laurent Binet, *La septième fonction du langage*

Finishing a dissertation is neither easy nor fast. While most of the work happened in the past ten years, the journey that led to this book really started thirty years ago. My mother explained me how to tell a story from all of the information I had read for my school presentations. My dad showed me the joys of programming and solving technical problems. Although I'll admit that the first program he showed me solved only simple arithmetic problems, and did not immediately impress me. Writing kept eluding me, despite my mom's best efforts. By the end of high school I would still panic when presented with a blank page. I had a long way to go. But I would not have been able to stand here today without what they had taught me.

I discovered my interest in programming languages in Eelco Visser's compiler construction courses. Without knowing it, I had dabbled in this area before. At my job at the time, I had written a program to generate installers for different platforms from a single XML description. I later tried a similar approach to generate code for MATLAB bindings of C libraries. "This should certainly save me time and make the process much more reliable than handwriting the bindings each time," I thought. The result was quite the opposite, because I didn't really know what I was doing. That changed when I discovered the wonderful world of programming languages. It started with compiler construction courses in Delft by Eelco and Guido, followed by a type theory course in Utrecht by Wouter Swierstra. What began as a few interesting coures, developed into a master thesis and then into an offer for a PhD position.

I am thankful to the many people I got to meet and work with over the course of my PhD. First of all Eelco. Without him none of this would have happened. His vision inspired the work I have done, and greatly influenced how I think about the development of programming languages. He taught me a lot about all aspects of research, and working with him has been a formative experience. It is a shame that he does not get to see more of his ideas come to fruition. I hope that my work and that of others after me can honor his vision and carry his ideas foward.

I want to thank Arie and Jesper for stepping in as promotor and copromotor. Your support and feedback has enabled me to finish this dissertation even after a delay of several years. I would also like to express my gratitude to the independent committee

members, Andrew Tolmach, Robby Findler, Görel Hedin, Wouter Swierstra, and Matthijs Spaan, for taking the time to assess my thesis, provide feedback, and be part of the defense.

I would like to extend many thanks to my collaborators: Andrew, for your continued involvement over the years, and Guido and Pierre, for your guidance in the early years. Arjen, I look back fondly on a stimulating and enjoyable time together, your critical approach and dry wit were always a pleasure. Casper, thanks for the good times in and outside the office, and for being a solid part of the team. Robbert, I appreciate that you would never let us cheat. Aron and Daniël, it was a pleasure working with you, and it is great to see in which directions you've taken this work.

I want to thank the Programming Languages research group in Delft, whom I have always found to be a kind and supportive group of people who just like to nerd out together. Vlad, I still miss your sarcasm at times. Peter, your feedback and improvement suggestions as an early adoptor of Statix have been truly valuable. Sebastian, your approach to research was an interesting counterpoint to Eelco's, and our midnight pub crawl in Florida was the best possible way to start conference life. Cas, Daco, Danny, Eduardo, Elmer, Gabriël, Jasper, Jeff, Jules, Michael, Paolo, Sven, and all the others, it has been a pleasure working with all of you. Martijn and Taico, supervising you was a nice and enriching experience. All the compiler construction students, thank you for being my guinea pigs. Your (sometimes forceful) feedback was invaluable for improving the practical aspects of Statix. A big thanks to Roniet for your support and the pleasant chats we had over the years.

I appreciate all the academics I met outside Delft, who shared some of their ideas or time with me. Jan Friso Groote, thank you for hosting me for a time at Eindhoven University of Technology. I regularly remember what you told me: if you cannot work out a solution, you have probably not understood the problem yet. John Hughes, many thanks for hosting me one summer at Chalmers University of Technology. I really enjoyed your delibrate style of working and developing ideas. Thank you Marianna, Matthew, and João, for being great conference buddies. Thanks to all the other students and professors I got to interact with at conferences, workshops, and summer schools.

I want to thank my colleagues at GitHub for the opportunity to continue working on code intelligence problems. It gave me a chance to learn what it takes to apply research ideas on a much larger scale. I want to thank my wonderful colleagues from the Semantic Code team. Thank you Doug, for welcoming me into the group, and for the many fun and interesting discussions about language design and the practical problems of implementing language tools. Rick, Rob, Patrick, Nina, and Beka, I had a lovely time working on code navigation together. Many thanks to my manager Terry for giving me the support and flexibility over the past year that I needed to be able to finish this disseration.

A thank you to all the great friends who have been there for me over the years: José, Chris, and Pia, I really value your support and good advice during trying times where I was ready to quit the whole thing. Koen and Aurinke, I am very happy and grateful for your support at my defense as my paranymphs. Rien and Hugo, thank you for enriching and having enriched my cultural and musical life. All my friends from SoSalsa, Monique, Vicki, Gerard, Guusje, Roos, Rostand, Ruben, Hiske, Asia, Jeanine, Erik, Vivian, Steven, Ruud, Milena, Nathan, KJ, and Angela, I love that we are still seeing each other after so many years. Mariana, Manu, and Koon, I really appreciate that, long after the days of Doerak and Bebop, we still get together occasionally. Natali, Danne, Valerie, Tom, Emma, Henry, and Sarah, the monthly no-birthday parties with you during the pandemic were a welcome diversion, and it is always nice to see you. My brothers, Dietger, Reinier, and Marten, whom I've known for ages, and my sisters-in-law, Krysia, Iris, and Celine, I appreciate you, and I'm happy I still get to know you better. Katia, Arthur, Fernando, Barbara, Vittorio, Nadia, Javanshir, and the rest of the poetry crowd, I am glad to have met you and hope for more good times together. To all the friends I forgot to mention here, apologies and thank you for being a part of my life.

It is well known that behind every successful student is a good barista. Thank you to Lucas and the others who tended the coffee cart downstairs in the faculty building, and to Sylwia and the staff of Harvest Cafe. It is great to be able to escape work in the afternoon and find a good coffee and a friendly chat waiting for you.

Finally I want to thank Nina. I am happy that you are a part of my life for so many years now. Without your support this dissertation would not exist. You encouraged me when I needed a push, and consoled me when I was completely through with all of it. Your feedback was sharp and to the point, and helped me many times improve the content or the process of my work. I could not be more thankful to have you by my side.

Hendrik van Antwerpen
Rotterdam, November 2024

Introduction

1

…, yet a computer program is surely an unsatisfying way to define semantics.

Donald E. Knuth

Name binding is an integral part of the static semantics of programming languages[1]. Common name binding features in the surface syntax of modern languages pose challenges for formal specifications as well as implementations. As a result, specifications are often restricted to a core language that lacks many of the surface language's features. Implementations require specialized data structures and must be structured specifically to support the full language. On top of that, name binding information is used beyond the compiler to support editor services. This dissertation proposes a meta-language for the specification of static semantics. It is based on scope graphs and supports the direct modeling of surface language name binding features. It stays close to a familiar inference-style of specification and allows deriving implementations for compilers and editor services.

1.1 Name Binding in Programming Languages

The development of programming languages began in earnest in the late 1940s and early 1950s, when the first programmable systems were developed. These systems allowed writing programs for a computer on that computer itself, and have them translated to low-level instructions by another program, the compiler. Computer programming was liberated from the low-level specifics of the computer it ran on, and programming languages became ″high-level″, offering constructs that abstracted over low-level instructions. (For a survey of those early developments, see Knuth and Pardo, 1980.)

Programming Environment An overview of modern programming environments is shown in Figure 1.1. A *parser* takes the source text and turns it into a structured abstract syntax tree (AST), according to the rules of the language grammar. The *checker* verifies that the program is well-formed, for example checking that names are bound and types are correct. These properties are described by the *static semantics* of a programming language, and ensure that certain kinds of errors cannot occur when the

[1] We will generally use "language" in the sense of programming languages. If another kind of language is meant, we will qualify it explicitly as, for example, "natural language".

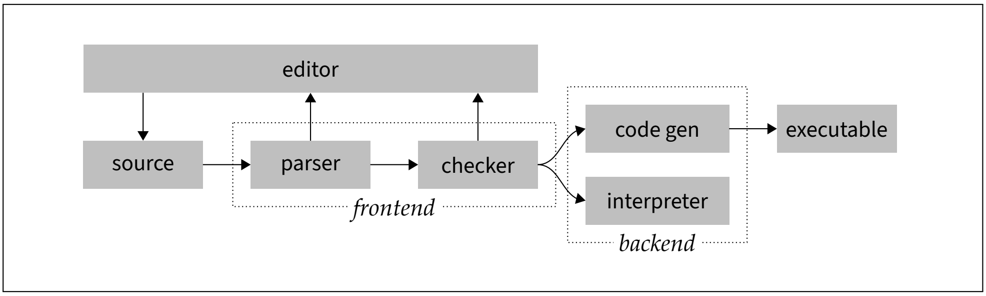


Figure 1.1: A schematic overview of the modern programming environment.

program is executed. The parser and checker together are usually called the frontend of the compiler. A *code generator* translates the program into an *executable* binary. Alternatively, the program is directly executed by an *interpreter*. The execution behavior of a programming language is described by its *dynamic semantics*. The compiler or interpreter are usually called a compiler's backend.

Besides this batch process of compilation, there is also interaction with the editor. The editor provides services based on information from the compiler frontend. One class of services helps users understand code by, for example, annotating it with inferred information, or supporting code navigation. Other services transform code, such as automatic refactoring and code repair. It is important to realize that these editors and editor services are often developed separately from the compiler as part of an integrated development environment (IDE). That means that static information, such as type and name binding, is no longer purely internal to the compiler.

```
// p/A.java
package p;
public class A {
  public int x;
}

// q/B.java
package q;
import p.*;
public class B {
  public int y;
  public int m(int y) {
    return new A().x + y;
  }
}
```

Figure 1.2: An example of name binding in Java.

Name Binding With the development of programming languages came the development of more complex name binding features. From variables, via subroutines, to packages, modules, and user-defined types, languages offered more and more possibilities for abstraction and ways to structure large programs. To make things more concrete, we illustrate name binding features commonly found in modern surface languages using an example, given in Figure 1.2.

Name binding describes the meaning of names in a program. Names are introduced by means of *declarations* and used through *references*. A reference is said to be *bound* to its declaration, and the process to determine these bindings is called *name resolution*. For example the reference y on line 13 is bound to the declaration on line 12. If a declaration is visible in a subtree of the AST we call it *lexically scoped*. An example is the parameter y on line 12, which is visible in the method body. Scoping is *non-lexical* if declarations are visible in other parts of the AST. Examples are the references A and x on line 13. The former is bound to the declaration A on line 3, which is exported as part of a named collection, the package p, and is visible because of the *import* on line 9. The latter is bound to the declaration x on line 4 through the expression type and is called *type-dependent*. Finally, if multiple declarations are available for a reference, *disambiguation* determines which declaration the reference is bound to. An example is the y reference on line 13, with possible declarations on lines 11 and 12, where the declaration on line 12 is said to *shadow* the one on line 11.

Name binding can be complex. To prove something about a program's behavior, or to implement a compiler or other tool for a language, we need more than illustrative examples. At that point, language specifications become important.

1.2 Programming Language Specifications

How can we give a description of what the names in a program, and the program source more generally, mean? As long as programming languages exist, people have produced descriptions of the meaning of programs in those languages. When these descriptions aim to give a precise and complete coverage of (part of) a language, we call them *specifications*. Specifications can be used for different purposes, and the format of the specification determines how suitable a specification is for a certain purpose. Therefore, before we discuss specifications in more detail, we should think about the possible uses that a specification may have:

- Explain the language to humans so they can understand and write programs in the language.
- Guide developers who implement tools such as compilers and IDEs for the language.
- Enable reasoning about properties of the language, its implementations, or programs in the language.
- Automatically derive implementations or proofs for the language.

Depending on the intended use, specifications may vary in formality, level of abstraction, and coverage. For example, a specification intended for people programming in the language should be high-level and abstract over details that may be essential for compiler engineers. Specifications intended for reasoning should be formal, while a specification for a compiler engineer could be less formal, although both should be precise and unambiguous. None of these differences make one specification better than another, as each has their own purpose. We now discuss two different specification styles and where they are positioned with respect to the previously discussed use cases.

Natural Language Specifications in a natural language have existed since the early days of programming languages. Examples are the reports on ALGOL (Backus et al., 1960; van Wijngaarden et al., 1976), and, more recently, specifications of Java (Gosling et al., 2018), C# (ECMA International, 2017), and Scala (Odersky et al., 2019). The use of natural language is particularly valuable for explaining the programming language to others. The introduction to the ALGOL 60 report states that the language in the report is guided by "ease of mutual understanding and not by computer limitations", and one of its explicit purposes is to be a "reference and guide for compiler builders." (Backus et al., 1960). However, making a specification in a natural language precise and unambiguous is challenging. The revised report on ALGOL 68 (van Wijngaarden et al., 1976) acknowledges this when it says that the "semantics [is] expressed in natural language, but making use of some carefully and precisely defined terms and concepts." Interestingly, they note "that this method maybe difficult for the uninitiated reader." The C# language specification, which is explicitly aimed at a broad audience of "implementers, academics, and application programmers", therefore compensates for the inaccessibility of a precise and technical description by including "a considerable amount of explanatory material that, strictly speaking, is not necessary in a formal language specification." A specification in natural language can be accessible, but making it precise can easily harm this accessibility, and result in large documents (the Java Language Specification comes in at over 750 pages). The lack of formality of natural language makes these specifications unfit for reasoning about the language, and requires additional formalization (and indeed, sometimes multiple such formalizations exist for various subsets of a language).

To illustrate how name binding can be specified in natural language, consider the following rule for Java:

> A declaration *d* of a local variable or exception parameter named *n* shadows, throughout the scope of *d*, (a) the declarations of any other fields named *n* that are in scope at the point where *d* occurs, and (b) the declarations of any other variables named *n* that are in scope at the point where *d* occurs but are not declared in the innermost class in which *d* is declared. (Gosling et al., 2018, p. 149)

The need for precision, and the fact that the rule is formulated in terms of concrete language constructs, makes it quite verbose. For a big surface language with many constructs to describe, this results in a specification lacking conciseness.

Formal Logic Language specifications using formal languages have existed almost as long as those in natural language. Building on the success of using the formalism of context-free grammars to describe syntax, researchers started looking for formalisms that allowed the description of semantics as well. The conference on *Formal Language Description Languages for Computer Programming* (T.B. Steel, 1966) was a seminal moment in the development of that field. (For a historical account of the development of programming semantics, see Astarte, 2019.) Compared to specifications in natural language, formal specifications tend to be more precise, more concise for their level of detail, and lend themselves better for reasoning. An example is the definition of Standard ML (Milner et al., 1997), which gives a precise definition of both static and dynamic semantics of the ML language. Its introduction explicitly dismisses the use of natural language as "ill-suited for use by an implementer, or by someone who wants to formulate laws for equivalence of programs, or by a programmer who wants to design programs with mathematical rigour."

Formally describing surface language name binding is challenging. Anything beyond basic lexical scoping does not fit the commonly used judgment[2] $\Gamma \vdash e : t$. First, the rules need to be restructured to ensure relevant non-lexical binding information is available in the context. The ML specification, for example, uses in- and output environments in several places, giving the rules a much more algorithmic flavour. Second, we need a representation for this richer context, together with rules for its usage. The authors of the ML specification discuss this point when they find a fictional language definer in the following situation:

> Here, he meets a problem; notation of some kind must be used to denote and describe these meanings —but not a programming language notation, unless he is passing the buck and defining one programming language in terms of another. Moreover, it is not enough just to write down mathematical definitions. The world of meanings only becomes meaningful if the objects possess nice properties, which make them tractable. [...] So the language-definer really has to develop a small theory of his meanings, in the same way that a mathematician develops a theory. [...] Of course he can take many objects and their theories directly from mathematics, such as functions, relations, trees, sequences, ... But he must also give some special theory for the objects which make his language partic-

[2] The judgement $\Gamma \vdash e : t$ states that an expression e has type t in a typing environment Γ. The typing environment Γ is a top-down constructed mapping from names to types, which means names only flow *downwards* to into subtrees.

> ular, as we do for types, structures and signatures in this book; otherwise his language definition may be formal but will give no insight. (Milner et al., 1997, p. xi)

This quote observes that not every formal language is necessarily suitable or useful to describe a programming language. The "small theory of his meanings" are the representations and abstractions that allow the specification to communicate the intent of the language and be useful for both understanding and theory development.

Between general mathematics and language specific theories are those that capture the commonalities between many programming languages. The use of such theories raises the level of abstraction of a specification, and promotes shared understanding by providing a common language to describe and compare different programming languages.

1.3 Meta-Languages

A specification does not automatically give us an implementation that makes a language usable in practice. Implementing a specification is a non-trivial and laborious task, for various reasons. First, a specification does not always cover the full language that is accepted by the compiler frontend. We call the language that is covered by the specification the *core* language, and the language that is accepted by the compiler the *surface* language. The latter is typically defined by its mapping into the core language. For example, the specification may always require explicit type annotations, while these are optional in the source and expected to be inferred. Second, high-level specifications can be very different from the algorithm that implements the specified behavior. An example is a constraint solving algorithm that implements a set of non-algorithmic subtyping rules. Third, a single implementation is not always enough to support all aspects of a development environment. The algorithm to support automatic refactoring is probably different from the algorithm that checks a complete program. Fourth, implementations are often specific to one language, requiring implementers to deal with low-level details that are common between languages. Ensuring that an implementation is correct is a non-trivial task, that requires extensive testing or proof work.

A class of meta-languages has been developed to address the challenges of high-level specifications, as well as the gap between specification and implementations. These meta-languages aim to provide reusable abstractions for common aspects of programming languages, as well as reusable implementations for specifications written in the meta-language. The design of meta-languages involves finding a balance between various aspects, such as what can be expressed, what details are abstracted away, and how the specification can be executed. An early example of a meta-language are the attribute grammars of Knuth (1968) that provide an abstraction to specify attributes of AST nodes by means of equations, which supports reasoning and allows an executable implementation as well. A meta-language allows language engineers to focus on the design and specification of a language, while greatly reducing the effort required to implement it. Furthermore, correctness has to be shown only once, instead of for each individual language implementation.

The development of interactive programming environments has led to the rise of language workbenches as platforms to develop programming languages (Fowler, 2005). Language workbenches offer an integrated programming environment for meta-languages, supporting meta-languages for different aspects of a programming language, and deriving not just static checkers or interpreters, but complete IDEs based on language specifications. Nowadays several mature language workbenches exist (Erdweg, van der Storm, Völter, et al., 2015), including the Spoofax Language Workbench (Kats and Visser, 2010), in the context of which the work in this dissertation has been developed.

1.4 Research Objective

The goal of this dissertation is to develop a meta-language for the specification of static semantics that (i) has a clear and clean underlying theory (*principled*); (ii) can handle a broad range of common language features (*expressive*); (iii) is *declarative*, but realizable by practical algorithms and tools (*executable*); (iv) is factored into language-specific and language-independent parts, to maximize reuse (*reusable*); and (v) can be applied to erroneous programs as well as to correct ones (*resilient*), goals previously outlined by Néron et al. (2015) and Zwaan and van Antwerpen (2023).

Our starting point is the theory of *scope graphs* (Néron et al., 2015), a language-independent model for name binding. Scope graphs model the binding structure of a program as a graph and define name resolution as a path finding problem in that graph, controlled by resolution policies that determine reachability and disambiguation. Scope graphs are formally defined by a resolution calculus and come with a resolution algorithm. Importantly, scope graphs were designed with our stated goals in mind. We now look at each of the goals in more detail and discuss what they mean for the design of our meta-language.

A *principled* design is based on a small set of, ideally orthogonal, concepts. It has a clear, formal description, gives unambiguous meaning to specifications written in the meta-language, and can be used to verify correctness of implementations.

An *expressive* design allows the specification of a wide range of surface language features. This means that lexical and non-lexical name binding patterns, as well as common type system features such as nominal and structural types, subtyping, and generics. The ability to model the interaction between names and types is important.

A *declarative* design allows specifications to be understood in terms of the possible models for a program, typically described by a declarative semantics, without consideration of a particular interpretation of the specification. The abstractions provided by the meta-language should match the domain, allow concise specifications, and avoid the need for cumbersome encodings. Specifically, we aim for a design that allows a specification to remain close to the familiar $\Gamma \vdash e : t$ judgments, without the need for the explicit bookkeeping that is typically associated with non-lexical name binding features.

An *executable* design allows for the specifications to be executed as type checkers or editor services. This means the language has an operational semantics and algorithms implementing it. While matching the performance of hand-written, optimized compilers is not a goal, performance should be good enough to allow these algorithms to be usable in practice. The declarative nature of the design allows for different interpretations beyond type checking, for example for code completion or program repair.

A *reusable* design clearly separates language-specific specification from generic, language-independent implementation. It also allows reuse of shared logic within and between specifications.

Finally, a *resilient* design is applicable to incorrect programs as well as correct ones. This means useful partial information is available in an interactive context. Additionally, partial solutions may be a starting point for developing autocompletion and program repair techniques.

These goals can easily conflict with each other. The more expressive the meta-language, the harder it can be to make it executable, and high-level abstractions can conflict with modeling the idiosyncrasies of some programming languages. The challenge is therefore to strike an interesting and useful balance between these goals. As such, the work we present is necessarily just one design point among many other possible designs. In Section 7.1 we return to these goals and consider to which extent the goals are realized by our design.

1.5 Research Method

The research in this dissertation is conducted through the design and implementation of a meta-language. The idea of this approach is that the meta-language is designed, implemented, and put in practice continuously. That means that all the different aspects of the meta-language are considered from the start to inform design choices and trade-offs. This leads to various artifacts and applications, which are presented in the following chapters.

First, we develop a formal description of the syntax and semantics (declarative and operational) of the meta-language. Writing these descriptions gives feedback on the complexity and theoretical elegance of the design. It helps us understand if the design is easy to understand and explain and if is it general enough. These descriptions are used to reason about the desired properties of the language. This helps us understand if the semantics are well defined, if the operational semantics are correct, confluent, and terminating.

Second, we implement the design to allow us and other users to experiment with the language. Writing an implementation is, first of all, a reality check. Every detail must be accounted for. One cannot rely on conventions or assumptions to make it work. An implementation is a valuable tool to increase understanding of the system's behavior. It gives the ability to experiment with different cases faster than working them out by hand. Benchmarks can be designed, both on artificial as well as actual code, to learn about the system's performance. Finally, an implementation makes the system available to other users, which is an important channel for feedback and new ideas.

Third, we conduct extensive case studies that test the design against existing, full-featured programming languages. The design process is typically driven by small, representative, examples, or difficult cases that are challenging for the state of the art. Developing extensive case studies mitigates the risk that the design is over-specialized for these initial cases. It forces us to consider many common cases, and shows how the meta-language design works for large languages and extensive specifications.

All the work is implemented as part of the Spoofax language workbench (Kats and Visser, 2010). Spoofax allows us to build on existing approaches for syntax specification and program transformations, as well as existing language specifications that have been developed in it. Providing the meta-language as part of a language workbench reduces the work required to integrate it in a complete compiler pipeline, and lowers the barrier of entry for other researchers, students, and industry.

1.6 Contributions

The work presented in this dissertation addresses the challenges of the previous section by developing a meta-language, Statix, for the specification of static semantics of programming languages. The goal of Statix is to provide first class support for high-level name binding concepts, while abstracting over operational concerns commonly associated with implementing name binding. In particular, this dissertation makes the following three high-level contributions:

1. We present a formal specification language, with a precise declarative semantics, for defining static semantics. It provides high-level abstractions for expressing name binding structure and name resolution queries.

2. We present an operational semantics for the meta-language that interprets specifications as type checkers. It abstracts over evaluation order and offers implicit parallelization.

3. We present case studies that show this approach is sufficient to express and interpret common name binding and type system features, and that it scales to the full surface syntax of real-world programming languages.

The chapters are divided into two parts. The main focus of the first part is specification, that of the second part is interpretation. We give a short introduction and summarize the main contributions of each chapter. All chapters have been published previously in peer reviewed venues and are included in their original form, except for formatting changes to fit this dissertation. Each chapter discusses contributions and related work in detail, and can be read independently. The chapters' core contributions are independent, but there is some overlap in motivation, background, examples, and discussion of related work. The original publication and the specific contributions by this dissertation's author for each chapter are mentioned below.

The specification chapters are concerned with meta-language design and its expressiveness for language specification. These chapters show the meta-language, its declarative semantics, and example specifications of programming languages in the meta-language:

- Chapter 2 presents a constraint language based on scope graphs. This is our first attempt at designing a meta-language for static semantics using scope graph concepts. Specifications are written as a straightforward mapping from an AST to a constraint term. The constraint language consists of term equality constraints, as well as scope graph assertions, and resolution constraints. We observe that the name binding model of scope graphs itself is not enough to express the resolution of all names. Some names (e.g., class members) depend on type information, preventing a strict separation of name resolution and type checking. Our approach interleaves name resolution and type checking, and supports limited interaction by allowing edge targets in the scope graph to be instantiated during analysis.

  Published as Hendrik van Antwerpen, Pierre Néron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth (2016). "A constraint language for static semantic analysis based on scope graphs." In: *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*. DOI: 10.1145/2847538.2847543.

  The author of this dissertation contributed substantially to the constraint language design, and carried out the entirety of the implementation. As customary in the research group, the underlying idea was developed jointly with the other authors and the publication was written in close collaboration with the co-authors.

- Chapter 3 presents the meta-language Statix, a logic language with first class support for scope graph assertions and queries. Specifications are written as predicates over the AST, but predicates are not limited to the AST and predicates over synthetic terms such as types are also allowed. We present a declarative semantics that describes the logical meaning of Statix programs. Compared to the previous constraint language, Statix is much more expressive, supporting the specification of, for example, type relations beyond equality such as structural typing, and polymorphism.

  While working on Statix we observed that the named import mechanism of Néron et al. (2015) could be expressed in Statix using a combination of queries and scope graph assertions depending on the query results. This sidesteps the possible exponential runtime of the original import mechanism, regularly triggered by users—even though it is always easy to mitigate. Therefore, named imports were dropped from Statix.

  Published as Hendrik van Antwerpen, Casper Bach Poulsen, Arjen Rouvoet, and Eelco Visser (2018). "Scopes as types." In: *Proceedings of the ACM on Programming Languages* OOPSLA. DOI: 10.1145/3276484.

  The author of this dissertation contributed substantially to the language design and carried out the entirety of the implementation. As customary in the research group, the underlying idea was developed jointly with the other authors and the publication was written in close collaboration with the co-authors.

The interpretation chapters are concerned with possible operational semantics for the meta-language:

- Chapter 4 presents an operational semantics that allows the interpretation of Statix programs as executable type checkers. This operational semantics abstracts over the evaluation order by dynamically scheduling predicate resolution based on instantiation of logical variables and scopes. Determining the completeness of scopes during resolution was the main challenge to ensure soundness. Our approach uses a combination of static restrictions on scope assertions with dynamic tracking of scope values to prevent the use of incomplete scope data and ensure stable query answers that cannot be invalidated by the remaining unresolved predicates. This makes the operational semantics incomplete with respect to the declarative semantics, but the various case studies have shown that these restrictions do not prevent us from specifying the languages we aim to support.

Published as Arjen Rouvoet, Hendrik van Antwerpen, Casper Bach Poulsen, Robbert Krebbers, and Eelco Visser (2020a). "Knowing when to ask: sound scheduling of name resolution in type checkers derived from declarative specifications." In: *Proceedings of the ACM on Programming Languages* OOPSLA. DOI: 10.1145/3428248.

The author of this dissertation contributed substantially to the idea and carried out the majority of the work for the case studies. The reformulation and formalization of Statix was a collaborative effort with the other authors, as customary in the research group. The proof and prototype implementation were mostly the first author's work. The publication writing was a joint effort.

- Chapter 5 presents a framework to implicitly parallelize scope-graph-based type checkers. Our approach exploits the fact that scope graphs provide a common representation of the binding structure to infer dependencies and decide scheduling of parallel type checkers. We apply this framework to Statix, which results in the implicit parallelization of all Statix specifications. Benchmark results show that significant scaling can be achieved without the effort necessary for tailored parallelization. Additionally, the specification for a significant subset of Java developed for this benchmark supports our claim that Statix is able to handle the surface syntax of a real programming language.

  Published as Hendrik van Antwerpen and Eelco Visser (2021). "Scope States: Guarding Safety of Name Resolution in Parallel Type Checkers." In: *35th European Conference on Object-Oriented Programming (ECOOP 2021).* Dagstuhl, Germany. DOI: 10.4230/LIPIcs.ECOOP.2021.1.

- Chapter 6 proposes to use Statix specifications for semantic editor services, such as code completion. It outlines an interpretation that would make this possible and discusses the challenges for realizing it. We argue that the name binding support of Statix allows a direct reinterpretation without the need, encountered by other approaches, to adapt the language specification itself.

  Published as Daniel A. A. Pelsmaeker, Hendrik van Antwerpen, and Eelco Visser (2019). "Towards Language-Parametric Semantic Editor Services Based on Declarative Type System Specifications (Brave New Idea Paper)." In: *33rd European Conference on Object-Oriented Programming (ECOOP 2019).* Dagstuhl, Germany. DOI: 10.4230/LIPIcs.ECOOP.2019.26.

  The first author of this publication and the author of this dissertation contributed equally to the idea and the writing of this work.

Finally, in Chapter 7 we draw conclusions and discuss future work.

All the work presented in this dissertation has been incorporated in the Spoofax Language Workbench (Kats and Visser, 2010). Downloads and documentation can be found at https://www.spoofax.dev/.

I

Specification

A Constraint Language for Static Semantic Analysis

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.

2.1 Introduction

Language workbenches (Erdweg, van der Storm, Völter, et al., 2015) 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 (Visser et al., 2014). 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 reuse; 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 (Néron et al., 2015). 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 iden-

Published as Hendrik van Antwerpen, Pierre Néron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth (2016). “A constraint language for static semantic analysis based on scope graphs.” In: *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*. DOI: 10.1145/2847538.2847543. Copyright © 2016 Owner/Author.

tifier 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 Néron et al. (2015) 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 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 name resolution relative to a scope graph. We define a constraint resolution

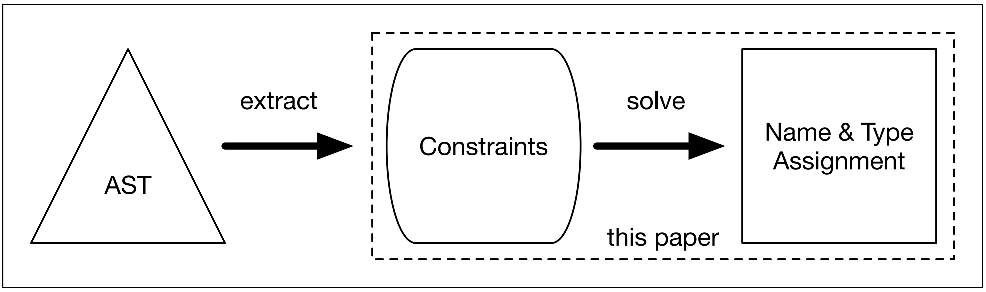


Figure 2.1: Architecture of our constraint-based approach to static semantic analysis. A language-specific extraction function translates an abstract syntax tree to a set of constraints. The generic constraint solver (independent of the source language), solves the constraints and produces a name and type assignment.

algorithm based on our previous name resolution algorithm, extended to support parameterization by a language-specific policy controlling scope reachability and visibility, combined with a standard unification algorithm. (iv) The constraint language is intended as an internal language for static semantic analysis tools (Fig. 2.1). Given the abstract syntax tree of a program, a language-specific extractor produces a set of constraints that express the name binding and types of the program. A language-independent solver attempts to find a solution for the set of extracted constraints, and produces a (partial) name and type assignment. Note that the constraint language is not intended as a domain-specific meta-language (such as NaBL; Konat et al., 2012) to be used by language designers using a language workbench. Rather, it is intended to be used as an internal language for the implementation of such meta-languages. (v) The application to erroneous programs is work in progress.

Contributions The specific technical contributions of this paper are the following:

- We introduce a constraint notation for the specification of scope graphs and name resolution that is complementary to the description of traditional typing constraints.

- We extend the scope graph framework of Néron et al. (2015) with uniqueness and completeness constraints to express properties such as "there are no duplicate declarations in this scope" or "every declared field in this record is initialized."

- We introduce generalized scope graph edge labels to model a wide range of scope combination policies including transitive and non-transitive imports, and non-overriding includes.

- We give a specification for satisfiability of combined sets of name and type resolution constraints.

- We extend the name resolution algorithm of Néron et al. (2015) to be parametric over scope reachability and visibility policies defined over (generalized) scope graph edge labels.

- We give an algorithm for solving combined name and type resolution problems and prove that it is sound with respect to the satisfiability specification.

Outline In Section 2.2, we introduce the constraint language using example programs in a small model language. In Section 2.3, we formally define the syntax and semantics of the constraint language by defining a satisfaction relation on constraints and an extended resolution calculus. In Section 2.4 we develop a constraint solver and prove that it is sound with respect to the semantics. In Section 2.5 we relate this work to previous work by ourselves and others, and discuss limitations and ideas for future work.

2.2 Constraints for Static Semantics

In this section we introduce our approach to constraint-based name and type resolution. We show how scope graph constraints are used to model name binding and combine them with typing constraints to model type consistency. We illustrate the ideas using LMR (Language with Modules and Records), a small model language that is a variant of the LM (Language with Modules) of Néron et al. (2015). LMR does not aspire to be a real programming language, but is designed to represent typical and challenging name and type resolution idioms.

In the rest of this section we study name and type resolution for a selection of LMR constructs using a series of examples. The full grammar of LMR is defined in Fig. 2.5 and a constraint extraction algorithm for the entire language is given in Fig. 2.6. Along the way we gradually introduce the concepts of the constraint language. The full syntax of the constraint language is defined in Fig. 2.7. Subsequent sections formalize the constraint language and its semantics.

2.2.1 Declarations and References

We first recall the concepts of the scope graph approach (Néron et al., 2015), and adapt them to a constraint-based framework. Consider the example in Fig. 2.2, which shows a simple LMR program with two global declarations (top), and, in the boxes below it, the constraints extracted from it and their solution. Subscripts on expressions and identifiers represent AST positions. Thus, x_1, x_4, and x_8 are different occurrences of the *same* name x. We represent scope graph constraints diagrammatically by the scope graph they specify.

The *nodes* of a scope graph $\mathcal{G}$ represent the three basic notions derived from the program abstract syntax tree (AST): *scopes, declarations, and references*:

- A *scope* is an abstraction of a set of nodes in the AST that behave uniformly with respect to name binding. Scopes are denoted by identifiers drawn from an abstract enumerable set. In a scope graph diagram, scopes are represented by circles with numbers representing their identity, e.g. $\textcircled{1}$. $\mathcal{S}(\mathcal{G})$ denotes the set of scopes of $\mathcal{G}$.

- A *declaration* is an occurrence of an identifier that introduces a name. We write x_i^{D} for the declaration of name x at position i in the program. We omit the position i when it is unimportant in the context. In diagrams, a declaration is represented by a box with an *incoming arrow*, e.g. $\rightarrow\boxed{\mathtt{x}_1}$. $\mathcal{D}(\mathcal{G})$ denotes the set of declarations of $\mathcal{G}$.

- A *reference* is an occurrence of an identifier referring to a declaration. We write x_i^{R} for a reference with name x at position i. Again, we sometimes omit the position i. In diagrams, a reference is represented by a box with an outgoing arrow, e.g. $\leftarrow\boxed{\mathtt{x}_4}$. $\mathcal{R}(\mathcal{G})$ denotes the set of references of $\mathcal{G}$.

Scope Graph Constraints The *edges* of a scope graph determine the connections between scopes, declarations, and references. Edges are specified directly by means of scope graph constraints (C^{G} in the grammar of Fig. 2.7), where the ground terms D, R, and S represent declarations, references, and scopes, respectively. For now, we only consider the two basic edges that connect declarations and references to scopes:

- A *declaration constraint* $s \longrightarrow x^{\mathsf{D}}$ specifies that declaration x^{D} belongs to scope s. Graphically: $\textcircled{s}\longrightarrow\boxed{x}$.

- A *reference constraint* $x^{\mathsf{R}} \longrightarrow s$ specifies that reference x^{R} belongs to scope s. Graphically: $\boxed{x}\longrightarrow\textcircled{s}$.

The "solution" to a set of scope graph constraints is a *well-formed* scope graph, i.e. one in which each declaration and reference belongs to (is connected by an edge with) exactly one scope. Note that the existence of nodes (declarations, references, and scopes) of the scope graph is specified implicitly by their appearance in an edge constraint. For convenience, we sometimes write $\mathcal{S}c(x^{\mathsf{D}}) = s$ for $s \longrightarrow x^{\mathsf{D}}$ and $\mathcal{S}c(x^{\mathsf{R}}) = s$ for $x^{\mathsf{R}} \longrightarrow s$. We define by comprehension the sets of declarations and references belonging to a scope s, as $\mathcal{D}(s) = \{x^{\mathsf{D}} \mid \mathcal{S}c(x^{\mathsf{D}}) = s\}$ and $\mathcal{R}(s) = \{x^{\mathsf{R}} \mid \mathcal{S}c(x^{\mathsf{R}}) = s\}$. In most contexts, constraints and derived notations are implicitly parameterized by the scope graph under consideration; when they need to be explicitly parameterized by a scope graph $\mathcal{G}$, we use a subscript notation (e.g. $\mathcal{D}_{\mathcal{G}}(s)$).

Resolution Constraints The basic semantic intuition behind scope graphs is that a reference resolves to a declaration iff there is a path from the reference node to the declaration node. In this case we say that the declaration is *visible* from the reference. Resolution constraints (C^{Res} in the grammar) represent requirements on successful name resolution:

- A *resolution constraint* $\mathsf{R} \mapsto \mathsf{D}$ specifies that a given reference must resolve to a given declaration. Typically, the declaration is specified as a declaration variable δ. For example, in Fig. 2.2 the constraints $\mathsf{x}_4^{\mathsf{R}} \mapsto \delta_4$ and $\mathsf{x}_8^{\mathsf{R}} \mapsto \delta_8$ require that references $\mathsf{x}_4^{\mathsf{R}}$ and $\mathsf{x}_8^{\mathsf{R}}$ resolve to (as yet unknown) declarations δ_4 and δ_8, respectively.

A solution to a set of resolution constraints is a substitution mapping each declaration variable to a declaration, such that applying this substitution to the constraints generates valid resolutions according to the scope graph resolution calculus (which we formalize in Section 2.3). In Fig. 2.2, since the only paths starting at $\mathsf{x}_4^{\mathsf{R}}$ and $\mathsf{x}_8^{\mathsf{R}}$ both end at declaration $\mathsf{x}_1^{\mathsf{D}}$, the (sole) solution to these constraints is a substitution mapping both δ_4 and δ_8 to $\mathsf{x}_1^{\mathsf{D}}$. Applying this substitution yields the valid resolutions $\mathsf{x}_4^{\mathsf{R}} \longmapsto \mathsf{x}_1^{\mathsf{D}}$ and $\mathsf{x}_8^{\mathsf{R}} \longmapsto \mathsf{x}_1^{\mathsf{D}}$.

In addition to constraints about the resolution of references, C^{Res} also includes constraints on properties of *name collections* N, which are multisets of identifiers. For now we only consider the uniqueness constraint:

- A *uniqueness constraint* !N specifies that a given name collection N contains no duplicates.
- A *declaration name collection* $\overline{\mathcal{D}}(s)$ is obtained by projecting the identifiers from the set of declarations in scope s.

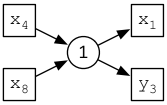
Thus, for example, in Fig. 2.2 the constraint $!\overline{\mathcal{D}}(1)$ requires that scope ① should have no duplicate declarations. These types of constraints are satisfied when the property they specify holds.

Typing Constraints Typing constraints (C^{Ty}) represent requirements for type consistency of the program:

- A *type declaration constraint* D : T associates a type with a declaration. This constraint is used in two flavors: associating a type variable (τ) with a concrete declaration, or associating a type variable with a declaration variable. In Fig. 2.2, the constraints $\mathsf{x}_1^{\mathsf{D}} : \tau_2$ and $\mathsf{y}_3^{\mathsf{D}} : \tau_9$ associate distinct type variables with declarations $\mathsf{x}_1^{\mathsf{D}}$ and $\mathsf{y}_3^{\mathsf{D}}$. (For ease of reading, we choose type variable names corresponding to subexpression label numbers.) The constraint $\delta_4 : \tau_4$ requires

$$\texttt{def}\ \mathtt{x}_1\ \texttt{=}\ \mathtt{1}_2$$
$$\texttt{def}\ \mathtt{y}_3\ \texttt{=}\ \texttt{(if (}\mathtt{x}_4\ \texttt{==}\ \mathtt{0}_5\texttt{)}_6\ \texttt{then}\ \mathtt{3}_7\ \texttt{else}\ \mathtt{x}_8\texttt{)}_9$$

Scope graph constraints



Resolution constraints

$$\begin{array}{c} \mathtt{x}_4^{\mathtt{R}} \mapsto \delta_4 \\ \mathtt{x}_8^{\mathtt{R}} \mapsto \delta_8 \\ !\overline{\mathcal{D}}(1) \end{array}$$

Typing constraints

$$\begin{array}{ll} \mathtt{x}_1^{\mathtt{D}} : \tau_2 & \tau_2 \equiv \mathit{Int} \\ \mathtt{y}_3^{\mathtt{D}} : \tau_9 & \tau_9 \equiv \tau_7 \\ \tau_9 \equiv \tau_8 & \delta_4 : \tau_4 \\ \delta_8 : \tau_8 & \tau_6 \equiv \mathit{Bool} \\ \tau_4 \equiv \mathit{Int} & \tau_5 \equiv \mathit{Int} \\ \tau_7 \equiv \mathit{Int} & \end{array}$$

Solution

$$\begin{array}{c} \delta_4 = \delta_8 = \mathtt{x}_1^{\mathtt{D}} \\ \tau_2 = \tau_4 = \tau_5 = \mathit{Int} \\ \tau_7 = \tau_8 = \tau_9 = \mathit{Int} \\ \tau_6 = \mathit{Bool} \end{array}$$

Figure 2.2: Declarations and references in global scope

the type of the declaration to which $\mathtt{x}_4^{\mathtt{R}}$ resolves to be the same as the type τ_4 of the reference considered as an expression.

- A *type equality constraint* $\mathrm{T} \equiv \mathrm{T}$ specifies that two types should be equal. In Fig. 2.2, the constraint $\tau_2 \equiv \mathit{Int}$ arises from the constant expression $\mathtt{1}_2$, and the constraint $\tau_4 \equiv \mathit{Int}$ arises from the fact that the `==` operator takes integer operands. The constraint $\tau_6 \equiv \mathit{Bool}$ arises in two ways, from the fact that `==` returns a Boolean and the fact that **`if`** requires one; since constraints should be thought of as a set, we list each distinct constraint only once.

A solution to a set of typing constraints is a substitution on declaration and type variables that satisfies all the constraints. For example, the substitution for τ_9 can be deduced either from the constraints $\tau_9 \equiv \tau_7$ and $\tau_7 \equiv \mathit{Int}$, or from the constraints $\tau_9 \equiv \tau_8$, $\tau_2 \equiv \mathit{Int}$ and the unification of τ_8 and τ_2 (via $\delta_8 = \mathtt{x}_1^{\mathtt{D}}$).

Note that for a program to be both well-bound and well-typed, we need to find a *single* substitution on declaration and type variables that allows both resolution and typing constraints to be satisfied simultaneously. In this simple example, it is clear that the declaration variables are determined solely by the resolution constraints, but this will not always be the case in general.

2.2.2 Lexical Scope

Only very trivial programs have just a single scope. The left part of Fig. 2.3 shows an LMR example that illustrates nested lexical scopes. Scope graphs use edges *between scopes* to model inclusion of the (visible) declarations in one scope in another. They can be used to model lexical nesting or direct import of all the names from one scope into another, according to the *label* on the edge.

- A *direct edge constraint* $s_1 \xrightarrow{l} s_2$ specifies a direct l-labeled edge from scope s_1 to s_2. (Graphically: $\textcircled{s_1} \xrightarrow{l} \textcircled{s_2}$.) The general meaning of such an edge is that the declarations visible in s_2 are also visible in s_1. Or, following the direction of the arrow, that a reference in s_1 can be resolved by searching for a declaration in s_2.

In the left part of Fig. 2.3, scope $\textcircled{2}$ — corresponding to the body of the `fun` — is nested within the program global scope $\textcircled{1}$, which is expressed by the scope edge constraint $\textcircled{2} \xrightarrow{\mathbf{P}} \textcircled{1}$. This edge is labeled **P** for "parent;" we will see other possible labels shortly. A resolution path starting from a reference may traverse a **P** edge to find a matching declaration, e.g. reference $\mathsf{f}^{\mathsf{R}}_{6}$ resolves to $\mathsf{f}^{\mathsf{D}}_{3}$. However, in order to model shadowing of outer declarations by inner ones, paths that traverse fewer (or no) **P** edges are preferred, so reference $\mathsf{n}^{\mathsf{R}}_{7}$ resolves to declaration $\mathsf{n}^{\mathsf{D}}_{4}$ rather than to $\mathsf{n}^{\mathsf{D}}_{1}$.

The kinds of typing constraints generated by this example are the same as those from the previous example. Note that the solution to the typing constraints leaves `f`'s result type unspecified (since it is never used).

2.2.3 Imports

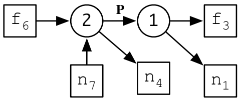
In addition to lexical scope, many programming languages provide features for making declarations in scopes selectively available 'at a distance'. Examples of such constructs are modules with imports in ML and classes with inheritance in Java. To model such features, scope graphs provide *associated scopes* and *imports*.

Associated Scope The essence of module-like constructs is that they encapsulate a collection of declarations and make these available through import of the module. That requires an association between the encapsulated declarations and the declaration of the module, which is modeled by *associated scopes*:

- An *association constraint* $x^{\mathsf{D}} \!-\!\!\!\triangleright s$ specifies s as the *associated scope* of declaration x^{D}. Associated scopes can be used to connect the declaration (e.g. a module) of a collection of names to the scope declaring those names (e.g. the body of a module). Graphically: $\boxed{x} \!-\!\!\!\triangleright \textcircled{s}$.

```
def n_1 = true_2
def f_3 = (
    fun (n_4:Int_5){
      f_6(n_7)
    }_8
)_9
```

Scope graph constraints



Resolution Constraints

$$\mathsf{f}^{\mathsf{R}}_6 \mapsto \delta_6 \qquad \mathsf{n}^{\mathsf{R}}_7 \mapsto \delta_7$$
$$!\overline{\mathcal{D}}(1) \qquad !\overline{\mathcal{D}}(2)$$

Typing constraints

$$\begin{array}{ll} \mathsf{n}^{\mathsf{D}}_1 : \tau_2 & \tau_2 \equiv \mathit{Bool} \\ \mathsf{f}^{\mathsf{D}}_3 : \tau_9 & \tau_9 \equiv \mathit{Fun}[\tau_5,\tau_8] \\ \mathsf{n}^{\mathsf{D}}_4 : \tau_5 & \tau_5 \equiv \mathit{Int} \\ \delta_6 : \tau_6 & \tau_6 \equiv \mathit{Fun}[\tau_7,\tau_8] \\ \delta_7 : \tau_7 & \end{array}$$

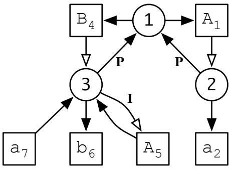
Solution

$$\begin{array}{ll} \delta_6 = \mathsf{f}^{\mathsf{D}}_3 & \delta_7 = \mathsf{n}^{\mathsf{D}}_4 \\ \tau_2 = \mathit{Bool} & \tau_8 = t_0 \end{array}$$
$$\tau_5 = \tau_7 = \mathit{Int}$$
$$\tau_6 = \tau_9 = \mathit{Fun}[\mathit{Int},t_0]$$

where t_0 is any fixed arbitrary type

```
module A_1 {
   def a_2 = 4_3
}
module B_4 {
  import A_5
  def b_6 = a_7
}
```

Scope graph constraints



Resolution constraints

$$\begin{array}{ll} \mathsf{a}^{\mathsf{R}}_7 \mapsto \delta_7 & !\overline{\mathcal{D}}(1) \\ !\overline{\mathcal{D}}(2) & !\overline{\mathcal{D}}(3) \end{array}$$

Typing constraints

$$\begin{array}{ll} \mathsf{a}^{\mathsf{D}}_2 : \tau_3 & \tau_3 \equiv \mathit{Int} \\ \mathsf{b}^{\mathsf{D}}_6 : \tau_7 & \delta_7 \ :\ \tau_7 \end{array}$$

Solution

$$\delta_7 = \mathsf{a}^{\mathsf{D}}_2$$
$$\tau_3 = \mathit{Int} \qquad \tau_7 = \mathit{Int}$$

Figure 2.3: Lexical scope and module imports.

The LMR program in the right part of Fig. 2.3 consists of two *modules* $\mathtt{A}_1$ and $\mathtt{B}_4$ and an import of the former into the latter. The declarations in these modules are contained in ② and ③ . Each of these scopes is *associated* with the corresponding declaration of the name of the module, which is represented in a scope graph diagram with an open arrow, e.g. $\boxed{\mathtt{A}_1} \mapsto$ ②. These scopes are also child scopes of the program global scope ① .

Imports A *nominal import* makes the declarations in an associated scope visible in another, not necessarily lexically related, *target scope*. A nominal import is represented by (1) a regular reference to the name of the scope being imported, and (2) an import edge of that name into the target scope:

- A *nominal edge constraint* $s \xrightarrow{l} x^{\mathsf{R}}$ specifies a nominal l-labeled edge from scope s to reference x^{R}. (Graphically: $\textcircled{s} \xrightarrow{l} \boxed{x}$) Such an edge makes visible in s all declarations that are visible in the associated scope of the declaration to which x^{R} resolves, according to the label on the edge.

For example, **`import`** $\mathtt{A}_5$ is represented by the reference $\mathtt{A}_5^{\mathsf{R}}$ in scope ③ and an import arrow ③$\xrightarrow{\mathrm{I}} \boxed{\mathtt{A}_5}$. It is also possible to import the declarations of another scope directly, using an (ordinary) nameless edge; this feature is used in the next sub-section.

Resolving through Imports Name resolution in the presence of associated scopes and imports proceeds as follows. If a scope S_1 contains an import $\mathtt{x}_i^{\mathsf{R}}$, which resolves to a declaration $\mathtt{x}_j^{\mathsf{D}}$ with associated scope S_2, then all declarations in S_2 are reachable in S_1. Thus, in the example, reference $\mathtt{a}_7^{\mathsf{R}}$ resolves to declaration $\mathtt{a}_2^{\mathsf{D}}$ since the import $\mathtt{A}_5^{\mathsf{R}}$ resolves to declaration $\mathtt{A}_1^{\mathsf{D}}$, and the associated scope ② of $\mathtt{A}_1^{\mathsf{D}}$ contains declaration $\mathtt{a}_2^{\mathsf{D}}$. Note that the resolution calculus is parameterized by the policy used to disambiguate conflicting resolutions. Here we use a default policy that prefers imported declarations over declarations in parents; alternatives are discussed in Section 2.3.4.

2.2.4 Type-Dependent Name Resolution

So far, we have seen how to use resolution constraints to express the dependence of type resolution on name resolution. However, for some language constructs the resolution of a name to its declaration depends on the type of another expression. For example, in a field access expression `e.f`, in order to resolve the field `f`, one first needs to find the type of the expression `e` and then to look for `f` in the scope associated with the type. This scheme induces dependencies on type resolution, not only from name resolution but also from scope graph construction (one does not know in which scope the reference `f` lies). We model such *type-dependent name resolution* by using scope graph constraints with scope variables. The examples in Fig. 2.4 illustrate the approach.

Field Declaration and Initialization Before we can study field access proper, we need to consider modeling of record types, field declarations, and record initialization. We identify each record type by the declaration of the record name in its type definition, e.g. $Rec(\mathsf{A}^{\mathsf{D}}_{1})$. We model the fields of a record type definition as declarations (here just $\mathsf{x}^{\mathsf{D}}_{2}$) in a scope (here, scope ②) associated with the record type name declaration $\mathsf{A}^{\mathsf{D}}_{1}$. The resolution constraint $!\overline{\mathcal{D}}(2)$ forbids duplicate field names.

To construct a new record of a declared record type (e.g. $\mathsf{A}^{\mathsf{D}}_{1}$), we create a new parentless scope (here, scope ③) which imports the field names of the record by importing (the associated scope of) the record declaration (via a reference to the name of the type, here $\mathsf{A}^{\mathsf{D}}_{5}$). We then process field initializers by putting references to the field names (here just $\mathsf{x}^{\mathsf{R}}_{6}$) into this new scope; these references can only resolve to the field declarations.

In order to check that each field of a record type is initialized, we use the following additional kinds of name collections and constraints:

- A *reference name collection* $\overline{\mathcal{R}}(s)$ denotes the multiset of reference identifiers of scope s.
- A *visible name collection* $\overline{\mathcal{V}}(s)$ denotes the multiset of declaration identifiers that are visible from scope s (i.e., would be visible from a reference to the declared identifier in s).
- A *subset constraint* $\mathrm{N} \lesssim \mathrm{N}$ specifies that one name collection is included in another.
- An *iso constraint* $\mathrm{N}_1 \approx \mathrm{N}_2$ is syntactic sugar for $\mathrm{N}_1 \lesssim \mathrm{N}_2 \wedge \mathrm{N}_2 \lesssim \mathrm{N}_1$ and specifies that two name collections are isomorphic.

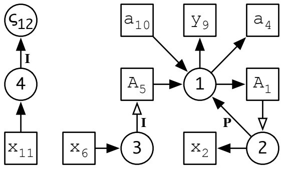
Thus, the constraint $\overline{\mathcal{V}}(3) \approx \overline{\mathcal{R}}(3)$ requires that the set of visible field declarations $\overline{\mathcal{V}}(3)$ (the declarations *visible* in scope ③) is isomorphic to the set of initializers $\overline{\mathcal{R}}(3)$ (the references in ③).

Field Access Now we consider the field access $\mathsf{a}^{\mathsf{R}}_{10}.\mathsf{x}^{\mathsf{R}}_{11}$ at subexpression 12 in Fig. 2.4. The reference $\mathsf{x}^{\mathsf{R}}_{11}$ is a field access in the record value of $\mathsf{a}^{\mathsf{R}}_{10}$. Thus, $\mathsf{x}^{\mathsf{R}}_{11}$ should be resolved in a scope containing (just) the declarations for the field names, i.e. the associated scope of the *type* of the $\mathsf{a}^{\mathsf{R}}_{10}$, namely ② . Once again, we create a parentless scope ④ and add the field being accessed (here $\mathsf{x}^{\mathsf{R}}_{8}$) as a reference in that scope. However, in this case we do not know at constraint extraction time that ② is the correct scope to import, because we do not know the type of $\mathsf{a}^{\mathsf{R}}_{10}$. *That is, the name resolution of x^{R}_{11} depends on the type resolution of a^{R}_{10}.*

To model this we proceed as follows. We create a new *scope variable* ς_{12} that acts as a placeholder for the scope that we want to import into scope ④. We add a direct

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

Scope graph constraints



Resolution constraints

$$\begin{array}{ll} \mathsf{A}^{\mathsf{R}}_{5} \mapsto \delta_8 & \mathsf{x}^{\mathsf{R}}_{6} \mapsto \delta_6 \\ \mathsf{a}^{\mathsf{R}}_{10} \mapsto \delta_{10} & \mathsf{x}^{\mathsf{R}}_{11} \mapsto \delta_{11} \\ !\overline{\mathcal{D}}(1) & !\overline{\mathcal{D}}(2) \\ \overline{\mathcal{V}}(3) \approx \overline{\mathcal{R}}(3) & \delta_{12} \rightsquigarrow \varsigma_{12} \end{array}$$

Typing constraints

$$\begin{array}{ll} \mathsf{x}^{\mathsf{D}}_{2} : \tau_3 & \tau_3 \equiv \mathit{Int} \\ \mathsf{a}^{\mathsf{D}}_{4} : \tau_8 & \tau_8 \equiv \mathit{Rec}(\delta_8) \\ \mathsf{y}^{\mathsf{D}}_{9} : \tau_{12} & \delta_6 : \tau_6 \\ \delta_{10} : \tau_{10} & \delta_{11} : \tau_{12} \\ \tau_7 \equiv \tau_6 & \tau_7 \equiv \mathit{Int} \\ & \tau_{10} \equiv \mathit{Rec}(\delta_{12}) \end{array}$$

Solution

$$\begin{array}{c} \delta_6 = \delta_{11} = \mathsf{x}^{\mathsf{D}}_{2} \\ \delta_8 = \delta_{12} = \mathsf{A}^{\mathsf{D}}_{1} \\ \delta_{10} = \mathsf{a}^{\mathsf{D}}_{4} \qquad \varsigma_{12} = 2 \\ \tau_3 = \tau_6 = \tau_7 = \tau_{12} = \mathit{Int} \\ \tau_8 = \tau_{10} = \mathit{Rec}(\mathsf{A}^{\mathsf{D}}_{1}) \end{array}$$

Figure 2.4: Field access.

edge constraint $\textcircled{4} \xrightarrow{\mathbf{I}} \textcircled{\varsigma_{12}}$, this time labeled with **I** rather than **P**, which makes the resolution process more eager to follow the edge (see Section 2.3.4 for details). We also have the usual constraints $a^{\mathsf{R}}_{10} \mapsto \delta_{10}$ and $\delta_{10} : \tau_{10}$ corresponding to reference $\mathsf{a}^{\mathsf{R}}_{10}$. And we have the constraint $\tau_{10} \equiv \mathit{Rec}(\delta_{12})$ for some unknown record type declaration δ_{12} because of the use of $\mathsf{a}^{\mathsf{R}}_{10}$ in the field position of a field access. To make the connection between the declaration of the record type and the placeholder scope, we use an association constraint:

- An *association constraint* $\mathrm{D} \rightsquigarrow \mathrm{S}$ specifies that a given declaration has a given associated scope.

Specifically, we use $\delta_{12} \rightsquigarrow \varsigma_{12}$ to say that ς_{12} must be the associated scope of δ_{12}.

Solving these constraints will lead to a solution for ς_{12} — in this case the associated scope of $\mathsf{A}^{\mathsf{D}}_{1}$, scope $\textcircled{2}$ — such that the appropriate scope can be imported into scope $\textcircled{4}$. After that, $\mathsf{x}^{\mathsf{R}}_{11}$ can be resolved as usual to the corresponding field declaration $\mathsf{x}^{\mathsf{D}}_{2}$, yielding its type $\tau_3 \equiv \mathit{Int}$.

With As a further variant, we discuss an expression form inspired by the `with` statement in the Pascal language. In the expression `with e do e'`, `e` should be a record-valued expression; the field names of the record are added to the lexical environment

```
 prog  =  decl*
 decl  =  module id{ decl*} | import id | def bind | record id{ fdecl*}
fdecl  =  id : ty
   ty  =  Int | Bool | id | ty → ty
  exp  =  int | true | false | id | exp ⊕ exp | if exp then exp else exp | fun ( id : ty ) { exp }
       |  exp exp | letrec tbind in exp | new id{ fbind*} | with exp do exp | exp . id
 bind  =  id = exp | tbind
tbind  =  id : ty = exp
fbind  =  id = exp
```

Figure 2.5: Syntax of LMR.

of `e'`. That is, a variable reference `x` in `e'` will be interpreted as a field of the record value when the record has indeed a field with name `x`; otherwise the variable is considered as a regular reference in the enclosing lexical context. Static resolution again requires resolving variables in `e'` in the associated scope of the record type of `e`, but this time also allowing resolution to the enclosing lexical scope. Replacing `(a.x)` by `(with a do x)` in the code of Fig. 2.4 produces identical constraints, with the addition of a scope graph edge $\textcircled{4} \xrightarrow{P} \textcircled{1}$.

This concludes the informal explanation-by-example of the constraint language and its application to LMR. A constraint extraction algorithm for the full LMR language is given in Fig. 2.6, but we do not discuss this in detail. Instead, in the next sections we formalize the syntax and semantics of the constraint language and discuss the definition of a resolution algorithm based on the semantics.

2.3 Syntax and Semantics of Constraints

In this section we formally define the syntax of the constraint language and its declarative semantics.

2.3.1 Syntax

Fig. 2.7 defines the full syntax of the constraint language. Constraints are divided into three categories: Scope graph constraints C^G specify a scope graph which defines the binding structures of the program. Resolution constraints C^{Res} describe requirements for all program names to be properly resolved and, where appropriate, to be unique or complete. Typing constraints C^{Ty} describe requirements for the program to be well-typed. The informal meaning of each constraint form was described by a bulleted definition in Section 2.2. Constraints can be combined using conjunction ($C \wedge C$) and True represents the trivially satisfiable constraint.

A *ground* constraint is one having no variables. A scope graph is *ground* if it is specified by a set of ground scope graph constraints; otherwise it is *incomplete*.

For simplicity, we describe the algorithm as operating over LMR's concrete syntax. The algorithm is defined by a family of functions indexed by syntactic category (*decl*, *exp*, etc.). Each function takes a syntactic component and possibly one or more auxiliary parameters, and returns a constraint, possibly involving one or more fresh variables or new scope identifiers. Functions are defined by a set of rules, one for each possible syntactic form in the category. For example, $\llbracket - \rrbracket^{exp}_{s,t}$ has twelve rules, and is parameterized by the scope s in which identifier references within the expression are to go and the expected type t of the expression. We use the notation $\llbracket - \rrbracket^{c^*}$ on sequences of items of syntactic category c to mean the result of applying $\llbracket - \rrbracket^{c}$ to each item and returning the conjunction of the resulting constraints, or True for the empty sequence.

$$
\begin{array}{rcl}
\llbracket ds \rrbracket^{prog} & := & !\overline{\mathcal{D}}(s) \;\wedge\; \llbracket ds \rrbracket^{decl^*}_{s} \\
\llbracket \texttt{module}\; x_i \,\{\; ds \} \rrbracket^{decl}_{s} & := & s \longrightarrow x^{\mathsf{D}}_i \;\wedge\; x^{\mathsf{D}}_i \mathrel{-\!\!\rhd} s' \;\wedge\; s' \xrightarrow{\mathbf{P}} s \wedge !\overline{\mathcal{D}}(s') \;\wedge\; \llbracket ds \rrbracket^{decl^*}_{s'} \\
\llbracket \texttt{import}\; x_i \rrbracket^{decl}_{s} & := & x^{\mathsf{R}}_i \longrightarrow s \;\wedge\; s \overset{\mathbf{I}}{\mathrel{-\!\!\rhd}} x^{\mathsf{R}}_i \\
\llbracket \texttt{def}\; b \rrbracket^{decl}_{s} & := & \llbracket b \rrbracket^{bind}_{s} \\
\llbracket \texttt{record}\; x_i \,\{\, fs \} \rrbracket^{decl}_{s} & := & s \longrightarrow x^{\mathsf{D}}_i \;\wedge\; x^{\mathsf{D}}_i \mathrel{-\!\!\rhd} s' \;\wedge\; s' \xrightarrow{\mathbf{P}} s \wedge !\overline{\mathcal{D}}(s') \;\wedge\; \llbracket fs \rrbracket^{fdecl^*}_{s,s'} \\
\llbracket x_i \texttt{ = } e \rrbracket^{bind}_{s} & := & s \longrightarrow x^{\mathsf{D}}_i \;\wedge\; x^{\mathsf{D}}_i : \tau \;\wedge\; \llbracket e \rrbracket^{exp}_{s,\tau} \\
\llbracket x_i \texttt{ : } t \texttt{ = } e \rrbracket^{bind}_{s} & := & s \longrightarrow x^{\mathsf{D}}_i \;\wedge\; x^{\mathsf{D}}_i : \tau \;\wedge\; \llbracket t \rrbracket^{ty}_{s,\tau} \;\wedge\; \llbracket e \rrbracket^{exp}_{s,\tau} \\
\llbracket x_i \texttt{:} t \rrbracket^{fdecl}_{s_r,s_d} & := & s_d \longrightarrow x^{\mathsf{D}}_i \;\wedge\; x^{\mathsf{D}}_i : \tau \;\wedge\; \llbracket t \rrbracket^{ty}_{s_r,\tau} \\
\llbracket \texttt{Int} \rrbracket^{ty}_{s,t} & := & t \equiv Int \\
\llbracket \texttt{Bool} \rrbracket^{ty}_{s,t} & := & t \equiv Bool \\
\llbracket t_1 \rightarrow t_2 \rrbracket^{ty}_{s,t} & := & t \equiv Fun[\tau_1,\tau_2] \;\wedge\; \llbracket t_1 \rrbracket^{ty}_{s,\tau_1} \;\wedge\; \llbracket t_2 \rrbracket^{ty}_{s,\tau_2} \\
\llbracket x_i \rrbracket^{ty}_{s,t} & := & t \equiv Rec(\delta) \;\wedge\; x^{\mathsf{R}}_i \longrightarrow s \;\wedge\; x^{\mathsf{R}}_i \mapsto \delta \\
\llbracket \texttt{fun}\; (x_i \texttt{:} t_1) \{\; e \} \rrbracket^{exp}_{s,t} & := & t \equiv Fun[\tau_1,\tau_2] \;\wedge\; s' \xrightarrow{\mathbf{P}} s \wedge !\overline{\mathcal{D}}(s') \;\wedge\; s' \longrightarrow x^{\mathsf{D}}_i \wedge\; x^{\mathsf{D}}_i : \tau_1 \\
 & & \qquad \wedge\; \llbracket t_1 \rrbracket^{ty}_{s,\tau_1} \;\wedge\; \llbracket e \rrbracket^{exp}_{s',\tau_2} \\
\llbracket \texttt{letrec}\; bs \;\texttt{in}\; e \rrbracket^{exp}_{s,t} & := & s' \xrightarrow{\mathbf{P}} s \wedge !\overline{\mathcal{D}}(s') \;\wedge\; \llbracket bs \rrbracket^{bind}_{s'} \;\wedge\; \llbracket e \rrbracket^{exp}_{s',t} \\
\llbracket n \rrbracket^{exp}_{s,t} & := & t \equiv Int \\
\llbracket \texttt{true} \rrbracket^{exp}_{s,t} & := & t \equiv Bool \\
\llbracket \texttt{false} \rrbracket^{exp}_{s,t} & := & t \equiv Bool \\
\llbracket e_1 \;\oplus\; e_2 \rrbracket^{exp}_{s,t} & := & t \equiv t_3 \;\wedge\; \tau_1 \equiv t_1 \;\wedge\; \tau_2 \equiv t_2 \;\wedge\; \llbracket e_1 \rrbracket^{exp}_{s,\tau_1} \;\wedge\; \llbracket e_2 \rrbracket^{exp}_{s,\tau_2} \\
 & & \qquad (\textit{where } \oplus \textit{ has type } t_1 \times t_2 \rightarrow t_3) \\
\llbracket \texttt{if}\; e_1 \;\texttt{then}\; e_2 \;\texttt{else}\; e_3 \rrbracket^{exp}_{s,t} & := & \tau_1 \equiv Bool \;\wedge\; \llbracket e_1 \rrbracket^{exp}_{s,\tau_1} \;\wedge\; \llbracket e_2 \rrbracket^{exp}_{s,t} \;\wedge\; \llbracket e_3 \rrbracket^{exp}_{s,t} \\
\llbracket x_i \rrbracket^{exp}_{s,t} & := & x^{\mathsf{R}}_i \longrightarrow s \;\wedge\; x^{\mathsf{R}}_i \mapsto \delta \;\wedge\; \delta : t \\
\llbracket e_1 \; e_2 \rrbracket^{exp}_{s,t} & := & \tau \equiv Fun[\tau_1,t] \wedge \llbracket e_1 \rrbracket^{exp}_{s,\tau} \wedge \llbracket e_2 \rrbracket^{exp}_{s,\tau_1} \\
\llbracket e \texttt{.} x_i \rrbracket^{exp}_{s,t} & := & \llbracket e \rrbracket^{exp}_{s,\tau} \;\wedge\; \tau \equiv Rec(\delta) \;\wedge\; \delta \rightsquigarrow \varsigma \;\wedge\; s' \xrightarrow{\mathbf{I}} \varsigma \;\wedge\; \llbracket x_i \rrbracket^{exp}_{s',t} \\
\llbracket \texttt{with}\; e_1 \;\texttt{do}\; e_2 \rrbracket^{exp}_{s,t} & := & \llbracket e_1 \rrbracket^{exp}_{s,\tau} \;\wedge\; \tau \equiv Rec(\delta) \;\wedge\; \delta \rightsquigarrow \varsigma \wedge\; s' \xrightarrow{\mathbf{P}} s \;\wedge\; s' \xrightarrow{\mathbf{I}} \varsigma \;\wedge\; \llbracket e_2 \rrbracket^{exp}_{s',t} \\
\llbracket \texttt{new}\; x_i \,\{\; bs \} \rrbracket^{exp}_{s,t} & := & x^{\mathsf{R}}_i \longrightarrow s \;\wedge\; x^{\mathsf{R}}_i \mapsto \delta \;\wedge\; s' \overset{\mathbf{I}}{\mathrel{-\!\!\rhd}} x^{\mathsf{R}}_i \wedge \llbracket bs \rrbracket^{fbind^*}_{s,s'} \;\wedge\; \overline{\mathcal{V}}(s') \approx \overline{\mathcal{R}}(s') \\
 & & \qquad \wedge\; t \equiv Rec(\delta) \\
\llbracket x_i \texttt{ = } e \rrbracket^{fbind}_{s,s'} & := & x^{\mathsf{R}}_i \longrightarrow s' \;\wedge\; x^{\mathsf{R}}_i \mapsto \delta \;\wedge\; \delta : \tau \;\wedge\; \llbracket e \rrbracket^{exp}_{s,\tau}
\end{array}
$$

Figure 2.6: Constraint extraction for LMR. Scope names (s) occurring free in rhs of rules are new ground scopes. Type variables (τ), scope variables (ς), and declaration variables (δ) occurring free are fresh variables.

$$
\begin{aligned}
\mathrm{C} &:= \mathrm{C}^{G} \mid \mathrm{C}^{Ty} \mid \mathrm{C}^{Res} \mid \mathrm{C} \wedge \mathrm{C} \mid \mathsf{True} \\
\mathrm{C}^{G} &:= \mathrm{R} \longrightarrow \mathrm{S} \mid \mathrm{S} \longrightarrow \mathrm{D} \mid \mathrm{S} \xrightarrow{l} \mathrm{S} \mid \mathrm{D} \rightarrowtriangle \mathrm{S} \mid \mathrm{S} \overset{l}{\rightarrowtriangle} \mathrm{R} \\
\mathrm{C}^{Res} &:= \mathrm{R} \mapsto \mathrm{D} \mid \mathrm{D} \rightsquigarrow \mathrm{S} \mid !\mathrm{N} \mid \mathrm{N} \lnsim \mathrm{N} \\
\mathrm{C}^{Ty} &:= \mathrm{T} \equiv \mathrm{T} \mid \mathrm{D} : \mathrm{T} \\
\mathrm{D} &:= \delta \mid x_i^{\mathsf{D}} \\
\mathrm{R} &:= x_i^{\mathsf{R}} \\
\mathrm{S} &:= \varsigma \mid n \\
\mathrm{T} &:= \tau \mid c(\mathrm{T}, ..., \mathrm{T}) \text{ with } c \in C_{\mathcal{T}} \\
\mathrm{N} &:= \overline{\mathcal{D}}(\mathrm{S}) \mid \overline{\mathcal{R}}(\mathrm{S}) \mid \overline{\mathcal{V}}(\mathrm{S})
\end{aligned}
$$

Figure 2.7: Syntax of constraints

The constraint language is parameterized by a family of type constructors $c \in C_{\mathcal{T}}$ and a set of labels $l \in \mathcal{L}$. We describe the former here and the latter in Section 2.3.4.

Type Constructors Types in T are either type variables τ or type constructor applications $c(\mathrm{T}, ..., \mathrm{T})$ with $c \in C_{\mathcal{T}}$, a set of language-specific type constructors. Each constructor c has an associated arity $c :: n$. For example, *Int* and *Bool* are type constructors with arity 0 and *Fun* is a type constructor with arity 2. Well-formed constraints respect the arity of the type constructors.

To represent user-defined types, such as classes in object-oriented languages or algebraic data types in functional languages, a type constructor can also include the scope graph declaration corresponding to the type definition. For example, record types in LMR are represented by $Rec(d)$ with d a type name declaration in the program; thus, in Fig. 2.4, the record definition `A` defines the type $Rec(\mathtt{A}_1^{\mathsf{D}})$.

2.3.2 Constraint Satisfaction

In our approach, the abstract syntax tree of a program p is reduced by the language-specific extraction function to a constraint $[\![p]\!] = \mathrm{C}_p^{G} \wedge \mathrm{C}_p^{Res} \wedge \mathrm{C}_p^{Ty}$ where commutativity and associativity of conjunction let us group the subconstraints into categories.

Our basic approach to defining satisfaction is as follows. First assume that we have only ground constraints. Then we can interpret scope graph constraints C^{G} directly as a ground scope graph. We next define a satisfiability relation $\models$ by cases on ground resolution constraints C^{Res} and typing constraints C^{Ty} relative to a context $(\mathcal{G}, \psi)$, where $\mathcal{G}$ is a ground scope graph and ψ is a typing environment mapping declarations in $\mathcal{D}(\mathcal{G})$ to unique ground types in T. In particular, resolution constraints are checked against $\mathcal{G}$ using the scope graph resolution calculus (described in Section 2.3.3). Finally, we apply $\models$ with $\mathcal{G}$ set to C^{G}.

$$\frac{}{\mathcal{G},\psi \models \mathsf{True}}\ (\textsc{C-True}) \qquad \frac{\mathcal{G},\psi \models C_1 \quad \mathcal{G},\psi \models C_2}{\mathcal{G},\psi \models C_1 \wedge C_2}\ (\textsc{C-And})$$

$$\frac{\psi(d) = T}{\mathcal{G},\psi \models d : T}\ (\textsc{C-TypeOf}) \qquad \frac{\vdash_{\mathcal{G}} p : x_i^{\mathsf{R}} \mapsto x_j^{\mathsf{D}}}{\mathcal{G},\psi \models x_i^{\mathsf{R}} \mapsto x_j^{\mathsf{D}}}\ (\textsc{C-Resolve})$$

$$\frac{d \longrightarrow\!\triangleright_{\mathcal{G}} S}{\mathcal{G},\psi \models d \rightsquigarrow S}\ (\textsc{C-ScopeOf}) \qquad \frac{\forall x, \mathbf{1}_{[\![N]\!]_{\mathcal{G}}}(x) \leq 1}{\mathcal{G},\psi \models !N}\ (\textsc{C-Unique})$$

$$\frac{[\![N_1]\!]_{\mathcal{G}} \subseteq [\![N_2]\!]_{\mathcal{G}}}{\mathcal{G},\psi \models N_1 \underset{\sim}{\subset} N_2}\ (\textsc{C-SubName}) \qquad \frac{t_1 = t_2}{\mathcal{G},\psi \models t_1 \equiv t_2}\ (\textsc{C-Eq})$$

Figure 2.8: Interpretation of resolution and typing constraints

To lift this approach to constraints with variables, we simply apply a multi-sorted substitution ϕ, mapping type variables τ to ground types, declaration variables δ to ground declarations and scope variables ς to ground scopes. Thus, our overall definition of satisfaction for a program p is:

$$\phi(\mathsf{C}_p^{G}), \psi \models \phi(\mathsf{C}_p^{Res}) \wedge \phi(\mathsf{C}_p^{Ty}) \qquad (\diamond)$$

where $\phi(E)$ denotes the application of the substitution ϕ to all the variables appearing in E that are in the domain of ϕ. When the proposition $\diamond$ holds we say that ψ and ϕ *resolve* p.

Resolution and Typing Constraints The $\models$ relation is given by the inductive rules in Fig. 2.8, where $=$ is the syntactic equality on terms and $\vdash_{\mathcal{G}} x_i^{\mathsf{R}} \longmapsto x_j^{\mathsf{D}}$ is the resolution relation for graph $\mathcal{G}$. The interpretation of a name collection $[\![\mathrm{N}]\!]_{\mathcal{G}}$ is the multiset defined as follows: $[\![\overline{\mathcal{D}}(S)]\!]_{\mathcal{G}} = \pi(\mathcal{D}_{\mathcal{G}}(S))$, $[\![\overline{\mathcal{R}}(S)]\!]_{\mathcal{G}} = \pi(\mathcal{R}_{\mathcal{G}}(S))$, and $[\![\overline{\mathcal{V}}(S)]\!]_{\mathcal{G}} = \pi(\{x_i^{\mathsf{D}} \mid \exists p, \vdash_{\mathcal{G}} p : S \longmapsto x_i^{\mathsf{D}}\})$ where $\pi(A)$ is the multiset produced by projecting the identifiers from a set A of references or declarations. Given a multiset M, $\mathbf{1}_M(x)$ denotes the multiplicity of x in M.

2.3.3 Resolution Calculus

The *resolution calculus* defines the *resolution* of a reference to a declaration in a scope graph as a *most specific, well-formed* path from reference to declaration through a sequence of edges. A path p is a list of steps representing the atomic scope transitions in the graph. There are three kinds of steps:

- A (direct) edge step $\mathbf{E}(l, S_2)$ is a direct transition from the current scope to the scope S_2. This step records the label of the scope transition that is used.

Resolution paths

$$\begin{array}{rcl} s & := & \mathbf{D}(x_i^{\mathsf{D}}) \mid \mathbf{E}(l, S) \mid \mathbf{N}(l, x_i^{\mathsf{R}}, S) \\ p & := & [\,] \mid s \mid p \cdot p \qquad \text{(inductively generated)} \end{array}$$

$$[\,] \cdot p = p \cdot [\,] = p$$

$$(p_1 \cdot p_2) \cdot\ p_3 = p_1 \cdot (p_2 \cdot p_3)$$

Well-formed paths

$$\mathit{WF}(p) \Leftrightarrow \mathit{labels}(p) \in \mathcal{E}$$

Visibility ordering on paths

$$\frac{\mathit{label}(s_1) < \mathit{label}(s_2)}{s_1 \cdot p_1 < s_2 \cdot p_2} \qquad \frac{p_1 < p_2}{s \cdot p_1 < s \cdot p_2}$$

Edges in scope graph

$$\frac{S_1 \xrightarrow{l} S_2}{\mathbb{I} \vdash \mathbf{E}(l, S_2) : S_1 \longrightarrow S_2}\ (E) \qquad \frac{S_1 \xrightarrow{l}\!\triangleright y_i^{\mathsf{R}} \quad y_i^{\mathsf{R}} \notin \mathbb{I} \quad \mathbb{I} \vdash p : y_i^{\mathsf{R}} \longmapsto y_j^{\mathsf{D}} \quad y_j^{\mathsf{D}} \!\longrightarrow\!\triangleright S_2}{\mathbb{I} \vdash \mathbf{N}(l, y_i^{\mathsf{R}}, S_2) : S_1 \longrightarrow S_2}\ (N)$$

Transitive closure

$$\frac{}{\mathbb{I}, \mathbb{S} \vdash [\,] : A \twoheadrightarrow A}\ (I) \qquad \frac{B \notin \mathbb{S} \quad \mathbb{I} \vdash s : A \longrightarrow B \quad \mathbb{I}, \{B\} \cup \mathbb{S} \vdash p : B \twoheadrightarrow C}{\mathbb{I}, \mathbb{S} \vdash s \cdot p : A \twoheadrightarrow C}\ (T)$$

Reachable declarations

$$\frac{\mathbb{I}, \{S\} \vdash p : S \twoheadrightarrow S' \quad \mathit{WF}(p) \quad S' \longrightarrow x_i^{\mathsf{D}}}{\mathbb{I} \vdash p \cdot \mathbf{D}(x_i^{\mathsf{D}}) : S \rightarrowtail x_i^{\mathsf{D}}}\ (R)$$

Visible declarations

$$\frac{\mathbb{I} \vdash p : S \rightarrowtail x_i^{\mathsf{D}} \quad \forall j, p' (\mathbb{I} \vdash p' : S \rightarrowtail x_j^{\mathsf{D}} \Rightarrow \neg(p' < p))}{\mathbb{I} \vdash p : S \longmapsto x_i^{\mathsf{D}}}\ (V)$$

Reference resolution

$$\frac{x_i^{\mathsf{R}} \longrightarrow S \quad \{x_i^{\mathsf{R}}\} \cup \mathbb{I} \vdash p : S \longmapsto x_j^{\mathsf{D}}}{\mathbb{I} \vdash p : x_i^{\mathsf{R}} \longmapsto x_j^{\mathsf{D}}}\ (X)$$

Figure 2.9: Resolution calculus of Néron et al. (2015) extended for arbitrary edge labels and parameterized with well-formedness predicate *WF* and visibility ordering $<$. Here *label* projects the label from a step and *labels* projects the sequence of labels from a path.

Lexical scope

$$\mathcal{L} := \{\mathbf{P}\} \quad \mathcal{E} := \mathbf{P}^* \quad \mathbf{D} < \mathbf{P}$$

Non-transitive imports

$$\mathcal{L} := \{\mathbf{P}, \mathbf{I}\} \quad \mathcal{E} := \mathbf{P}^* \cdot \mathbf{I}^? \quad \mathbf{D} < \mathbf{P}, \quad \mathbf{D} < \mathbf{I}, \quad \mathbf{I} < \mathbf{P}$$

Transitive imports

$$\mathcal{L} := \{\mathbf{P}, \mathbf{TI}\} \quad \mathcal{E} := \mathbf{P}^* \cdot \mathbf{TI}^* \quad \mathbf{D} < \mathbf{P}, \quad \mathbf{D} < \mathbf{TI}, \quad \mathbf{TI} < \mathbf{P}$$

Transitive Includes

$$\mathcal{L} := \{\mathbf{P}, \mathbf{Inc}\} \quad \mathcal{E} := \mathbf{P}^* \cdot \mathbf{Inc}^* \quad \mathbf{D} < \mathbf{P}, \quad \mathbf{Inc} < \mathbf{P}$$

Transitive includes and imports, and non-transitive imports

$$\mathcal{L} := \{\mathbf{P}, \mathbf{Inc}, \mathbf{TI}, \mathbf{I}\} \quad \mathcal{E} := \mathbf{P}^* \cdot (\mathbf{Inc} \mid \mathbf{TI})^* \cdot \mathbf{I}^?$$
$$\mathbf{D} < \mathbf{P}, \quad \mathbf{D} < \mathbf{TI}, \quad \mathbf{TI} < \mathbf{P}, \quad \mathbf{Inc} < \mathbf{P}, \quad \mathbf{D} < \mathbf{I}, \quad \mathbf{I} < \mathbf{P},$$

Figure 2.10: Example reachability and visibility policies by instantiation of path well-formedness and visibility.

- A nominal edge step $\mathbf{N}(l, y^{\mathsf{R}}, S)$ requires the resolution of reference y^{R} to a declaration with associated scope S to allow a transition between the current scope and scope S.
- A complete path always ends with a declaration step $\mathbf{D}(x^{\mathsf{D}})$ that stores the declaration the path is leading to.

A path p is a valid resolution in the graph from reference x_i^{R} to declaration x_i^{D} such that $\vdash_{\mathcal{G}} p : x_i^{\mathsf{R}} \longmapsto x_i^{\mathsf{D}}$ according to the calculus rules in Fig. 2.9. These rules all implicitly apply to a fixed graph $\mathcal{G}$, which we omit to avoid clutter. The calculus defines the resolution relation in terms of edges in the scope graph, reachable declarations, and visible declarations. Here $\mathbb{I}$ is the set of *seen imports*, a technical device needed to avoid "out of thin air" anomalies in resolution of nominal imports. We often drop $\mathbb{I}$ from a resolution when it is empty. The $\mathbb{S}$ component that appears in the transitive closure rules is the set of *seen scopes* that is used to prevent cycles in the resolution path of a given reference.

2.3.4 Parameterization

In order to model the name binding features and resolution policies from different programming languages, the scope graph and resolution calculus are parameterized with a set of labels $\mathcal{L}$, a regular expression $\mathcal{E}$ that defines the scope reachability policy, and an order $<$ on the $\mathcal{L}$ (extended with the built-in $\mathbf{D}$ label) that defines the scope visibility policy. Fig. 2.9 defines generic predicates derived from these parameters and used in the calculus. The regular expression $\mathcal{E}$ entails a *well-formedness predicate WF* on paths obtained by projecting the sequence of labels from the path and testing it for membership in the language of $\mathcal{E}$. The ordering relation on labels entails an ordering relation on paths using the lexicographic order on the projected label sequences.

Fig. 2.10 presents several example instantiations for these parameters, encoding different policies. The first policy defines lexical scope in which scopes are transitively linked through parent edges (**P**) and local declarations shadow declarations in parents. The next policy extends lexical scope with non-transitive imports (**I**). The well-formedness predicate allows an optional import at the end of a lexical scope chain, ruling out access to the parents of an imported scope. Further, the policy states that imported declarations shadow declarations in the lexical context. The transitive imports policy extends this by allowing paths with a chain of imports (**TI**). The transitive includes policy is a variation in which local declarations do not shadow included (**Inc**) declarations. The final policy combines three import policies, not providing rules to disambiguate between paths through different kinds of import edges. Thus, a reference that can be resolved through an import *and* an include edge is ambiguous and can be flagged as an error.

2.4 Resolution Algorithm

In this section, we describe an algorithm for solving constraints in the sense of Section 2.3.2, i.e. finding ϕ and ψ that satisfy ($\diamond$). Our algorithm works only for a restricted class of generated constraints: all constraints in C_p^G must be ground, except that scope variables ς can appear as targets in direct edge constraints (e.g. $S \xrightarrow{l} \varsigma$). This restriction is met by the constraints generated by the LMR collection algorithm in Section 2.2. Broader classes of constraints might be useful for other languages; we defer exploration of algorithms that could handle these to future work.

2.4.1 Variables in Scope Graph Constraints

The basic approach of the algorithm is to interpret the scope graph constraints as a scope graph $\mathcal{G}$ and then use it to resolve resolution and typing constraints using a conventional unification-based algorithm. However, since scope graph constraints can contain variables, we cannot fully define the scope graph before starting constraint resolution, because we do not fully know ϕ. Thus, our algorithm builds ϕ (and Ψ) incrementally. The key idea is that we can solve some resolution and typing constraints even when ϕ is not yet fully defined, in such a way that the solution remains valid as it becomes more defined.

$$
\begin{aligned}
R[\mathbb{I}](x^{\mathsf{R}}) &:= \text{let } (r,s) = Env_{\mathcal{E}}[\{x^{\mathsf{R}}\} \cup \mathbb{I}, \emptyset](\mathcal{S}c(x^{\mathsf{R}}))\} \text{ in} \\
&\quad \begin{cases} \mathsf{U} \quad \text{if } r = \mathsf{P} \text{ and } \{x^{\mathsf{D}} | x^{\mathsf{D}} \in s\} = \emptyset \\ \{x^{\mathsf{D}} | x^{\mathsf{D}} \in s\} \end{cases} \\
Env_{re}[\mathbb{I},\mathbb{S}](S) &:= \begin{cases} (\mathsf{T},\ \emptyset) \quad \text{if } S \in \mathbb{S} \text{ or } re = \varnothing \\ Env_{re}^{\mathcal{L} \cup \{\mathbf{D}\}}[\mathbb{I},\mathbb{S}](S) \end{cases} \\
Env_{re}^{L}[\mathbb{I},\mathbb{S}](S) &:= \bigcup_{l \in Max(L)} \left(Env_{re}^{\{l' \in L | l' < l\}}[\mathbb{I},\mathbb{S}](S) \lhd Env_{re}^{l}[\mathbb{I},\mathbb{S}](S) \right) \\
Env_{re}^{\mathbf{D}}[\mathbb{I},\mathbb{S}](S) &:= \begin{cases} (\mathsf{T},\ \emptyset) \quad \text{if } [\,] \notin re \\ (\mathsf{T},\ \mathcal{D}(S)) \end{cases} \\
Env_{re}^{l}[\mathbb{I},\mathbb{S}](S) &:= \begin{cases} (\mathsf{P},\ \emptyset) \ \text{if } S_l^{\blacktriangleright} \text{ contains a variable or } IS^l[\mathbb{I}](S) = \mathsf{U} \\ \displaystyle\bigcup_{S' \in \left(IS^l[\mathbb{I}](S) \cup S_l^{\blacktriangleright}\right)} Env_{(l^{-1}re)}[\mathbb{I}, \{S\} \cup \mathbb{S}](S') \end{cases} \\
IS^l[\mathbb{I}](S) &:= \begin{cases} \mathsf{U} \quad \text{if } \exists y^{\mathsf{R}} \in (S_l^{\triangleright} \backslash \mathbb{I}) \ s.t.\ R[\mathbb{I}](y^{\mathsf{R}}) = \mathsf{U} \\ \{S' \mid y^{\mathsf{R}} \in (S_l^{\triangleright} \backslash \mathbb{I}) \wedge y^{\mathsf{D}} \in R[\mathbb{I}](y^{\mathsf{R}}) \wedge y^{\mathsf{D}} \relbar\!\!\rhd S'\} \end{cases}
\end{aligned}
$$

Figure 2.11: Name Resolution Algorithm

2.4.2 Name Resolution Algorithm

In order to solve resolution constraints (e.g. $x^{\mathsf{R}} \mapsto \delta$) or to compute the set of visible elements from a scope ($\mathcal{V}(S)$) we need an algorithm that computes the name resolution relation ($x_i^{\mathsf{R}} \longmapsto x_j^{\mathsf{D}}$) specified by the calculus presented in Section 2.3.3. We introduced such an algorithm in our prior work (Néron et al., 2015), but it was specific to a particular set of labels, visibility order, and well-formedness predicate. In this section, we present a generic version of the algorithm that is parameterized by $\mathcal{L}$, $\mathcal{E}$ and $<$ as described in Section 2.3.4.

Incomplete Scope Graphs A further new requirement on the algorithm is that it can operate on an incomplete scope graph, specified by a set of constraints that may still contains variables as the targets of direct edges. The non-strictly positive premise of the (V) rule of the resolution calculus makes the derivation of a resolution relation from a graph non-monotonic with respect to additions to the graph. For example, suppose that in some graph $\mathcal{G}$ a reference x^{R} in a scope S resolves to declaration x_i^{D} in the parent scope S'. In a bigger graph $\mathcal{G}'$ that also has a declaration $x_{i'}^{\mathsf{D}}$ in S itself, x^{R} will resolve to $x_{i'}^{\mathsf{D}}$, and the old resolution to x_i^{D} will be shadowed. Thus we cannot simply restrict resolution to the complete part of the graph, and expect the results to remain valid as the graph becomes more completely known. Instead, we modify the original algorithm to signal when a result is preliminary.

The Algorithm Fig. 2.11 defines a resolution algorithm that works on such incomplete scope graphs.The function for resolving a single reference, $R[\mathbb{I}](x^{\mathsf{R}})$, returns either a set of declarations or U (*unknown*) if the reference cannot be resolved in the current graph. Similarly, the environment functions $Env_{\overline{re}}[\mathbb{I},\mathbb{S}](S)$ return a pair consisting of:

- a result flag, T (*total*) if all declarations visible from S can be computed or P (*partial*) if there are still possible additional resolutions (some scope variables are accessible)
- a set of declarations corresponding to resolutions from scope S that are already certain in this incomplete graph.

When a scope graph contains no variables (i.e. when no partial or unknown flags are raised) the intended behavior of the different functions is the following:

- $R[\mathbb{I}](x^{\mathsf{R}})$ returns the set of declarations to which the reference resolves.
- $Env_{re}[\mathbb{I},\mathbb{S}](S)$ returns the set of declarations that are reachable from scope S with a minimal path satisfying the regular expression *re*.
- $Env_{re}^{L}[\mathbb{I},\mathbb{S}](S)$ returns the set of declarations visible from S through labels in set L after application of the shadowing policy. Using the label order, the declarations accessible through smaller labels shadow the declarations accessible through larger ones.
- $Env_{re}^{\mathsf{D}}[\mathbb{I},\mathbb{S}](S)$ returns the set of declarations accessible from S with a **D** step, i.e. the set of declarations in S.
- $Env_{re}^{l}[\mathbb{I},\mathbb{S}](S)$ returns the set of declarations accessible from S with an l-labeled step.
- $IS^{l}[\mathbb{I}](S)$ returns the set of scopes that are accessible through a *nominal edge* by resolving the reference and returning its associated scope.

The algorithm uses the following auxiliary notation and definitions: $\varnothing$ denotes the empty regular expression and given a path p and a regular expression *re*, $p \in re$ denotes that $labels(p)$ is in the language of *re*. The shadowing operator $\lhd$ on sets of declarations is defined by:

$$D_1 \lhd D_2 \triangleq \left\{ x_i^{\mathsf{D}} \mid x_i^{\mathsf{D}} \in D_1 \lor (x_i^{\mathsf{D}} \in D_2 \land \nexists j, x_j^{\mathsf{D}} \in D_1 \right\}.$$

The shadowing operators on pairs with result flag are defined by:

$$(f_1, D_1) \lhd (f_2, D_2) \triangleq \begin{cases} (f_2, D_1 \lhd D_2) & \text{if } f_1 = \mathsf{T} \\ (\mathsf{P}, D_1) & \text{otherwise} \end{cases}$$

The union $\cup$ operator over pairs with result flag is defined as:

$$\bigcup_{i \in I} (f_i, D_i) \triangleq \begin{cases} (\mathsf{T}, D) & \text{if } \forall i \in I, (f_i = \mathsf{T}) \\ (\mathsf{P}, D) & \text{otherwise} \end{cases}$$

where $D = \left\{ x^{\mathsf{D}} \in \cup_{i \in I} D_i \mid (\forall j \in I, f_j = \mathsf{T} \vee \exists x_k^{\mathsf{D}} \in D_j) \right\}$.
Given a regular expression over labels *re* and a label l, $l^{-1}re$ denotes the Brzozowski derivative (Brzozowski, 1964) of *re* by l. Given a partially ordered set L, $Max(L)$ denotes the set of maximal elements of L, i.e. $\{l \in L \mid \nexists l' \in L, l < l'\}$. Given a scope S and a label l, we define:

$$S_l^{\triangleright} \triangleq \{x^{\mathsf{R}} \mid S \xrightarrow{l}\!\triangleright\, x^{\mathsf{R}}\} \qquad S_l^{\blacktriangleright} \triangleq \{S' \mid S \xrightarrow{l}\!\blacktriangleright\, S'\}$$

2.4.3 Correctness

We want to prove the correctness of this algorithm with respect to the calculus introduced in Section 2.3.3. Details of the proofs can be found in the appendix of the extended version (van Antwerpen et al., 2015).

Termination First notice that the algorithm terminates using the lexicographic ordering $(\#(\mathcal{R}(\mathcal{G}) \backslash \mathbb{I}), \#(\mathcal{S}(G) \backslash \mathbb{S}), \mathcal{O})$, where $\#(A)$ denotes the cardinality of set A and $\mathcal{O}$ is the following well founded order among the different functions:

$$Env_{re} > Env_{re}^{L} > Env_{re}^{l} > Env_{re}^{D} > IS > R$$

This termination order is used as the induction principle in most of the proofs.

Correctness on ground scope graphs We want to prove that when this algorithm operates on a ground scope graph, it is sound and complete with respect to the calculus presented in Fig. 2.9. First, it is trivial to prove that on a ground scope graph, the return flag can never be P or U, therefore in this section we forget about the flag and assume that the *Env* functions return a set of declarations.

To prove the correctness of the algorithm, we consider the set of paths that corresponds to the sets of declarations returned by the different functions. Given two sets of scopes $\mathbb{I}$ and $\mathbb{S}$ and a scope S, we define $\mathbb{P}[\mathbb{I}, \mathbb{S}](S)$ as:

$$\{p \cdot \mathbf{D}(d) \mid \exists S', \mathbb{I}, \mathbb{S} \cup \{S\} \vdash p : S \twoheadrightarrow S' \wedge \mathcal{S}c(d) = S'\}$$

and given a path p such that $p = p' \cdot \mathbf{D}(d)$, $\Delta(p)$ denotes the declaration d. For a set of paths S, $\Delta(S)$ denotes its corresponding set of declarations $\{\Delta(p) \mid p \in S\}$ and

$$\lhd S \triangleq \{\, p \cdot \mathbf{D}(x_i^{\mathsf{D}}) \in S \mid \forall\, (p' \cdot \mathbf{D}(x_j^{\mathsf{D}})) \in S, \neg\, p' < p\}$$

Given these definitions, we can state the correctness of the algorithm:

Lemma 1 (Resolution algorithm correctness). On a ground scope graph, we have the following equivalences

$$\begin{aligned}
R[\mathbb{I}](x^{\mathsf{R}}) =& \Delta(\{p \mid \exists d,\ \mathbb{I} \vdash p : x^{\mathsf{R}} \longmapsto d\}) \\
Env_{re}[\mathbb{I},\mathbb{S}](S) =& \begin{cases} \emptyset & \text{if } S \in \mathbb{S} \\ \Delta(\lhd\{p \cdot \mathbf{D}(d) \in \mathbb{P}[\mathbb{I},\mathbb{S}](S) \mid p \in re\}) & \text{otherwise} \end{cases} \\
Env_{re}^{L}[\mathbb{I},\mathbb{S}](S) =& \Delta(\lhd\{p \mid \exists l \in L, p \in \mathbb{P}_{re}^{l}[\mathbb{I},\mathbb{S}](S)\}) \\
Env_{re}^{\mathsf{D}}[\mathbb{I},\mathbb{S}](S) =& \Delta(\{\mathbf{D}(d) \mid [] \in re \wedge \mathcal{S}c(d) = S\}) \\
Env_{re}^{l}[\mathbb{I},\mathbb{S}](S) =& \Delta\left(\left\{ s \cdot p \;\middle|\; \begin{array}{l} label(s) = l \wedge \\ \mathbb{I} \vdash s : S \longrightarrow S' \wedge \\ p \in \mathbb{P}_{l^{-1}re}[\mathbb{I},\mathbb{S} \cup \{S\}](S') \end{array} \right\}\right) \\
IS^{l}[\mathbb{I}](S) =& \{S' \mid \exists y^{\mathsf{R}}, \mathbb{I} \vdash \mathbf{N}(l, y^{\mathsf{R}}, S') : S \longrightarrow S'\}
\end{aligned}$$

Proof. The proof is by induction on the termination order of the algorithm. Key observations are that all the considered sets of paths are finite since all the paths are acyclic and if there is a minimal path $s \cdot p$ from scope S with $\mathbb{I} \vdash s : S \longrightarrow S'$ then its tail p is also minimal from S', due to the lexicographic ordering. □

Correctness on incomplete scope graphs We now want to state the general correctness of the algorithm that can operate on incomplete scope graphs. We first extend this definition of resolution as follows. Given an incomplete scope graph $\mathcal{G}$, a reference x^{R} is said to resolve to a declaration x_i^{D} if and only if this resolution is valid in all ground instances of $\mathcal{G}$:

$$\vdash_{\mathcal{G}} x^{\mathsf{R}} \longmapsto x_i^{\mathsf{D}} \triangleq \forall\, \phi,\ \vdash_{\phi(\mathcal{G})} x^{\mathsf{R}} \longmapsto x_i^{\mathsf{D}} \qquad (\blacklozenge)$$

where we write $\vdash_{\mathcal{G}}$ for the resolution relation for graph $\mathcal{G}$ and $\phi(G)$ is the ground scope graph corresponding to the application of substitution ϕ to variables in $\mathcal{G}$. Similarly a declaration x^{D} is visible from scope S in an incomplete scope graph $\mathcal{G}$ if and only if it is visible in all the ground instances.

In order to be able to resolve uniqueness constraints for a program we also want to ensure that an incomplete graph provides *all* the possible resolutions of a given reference. In particular, if a resolution is unique in an incomplete graph, we want to be sure it is unique in all its ground instances. An incomplete graph $\mathcal{G}$ is *stable* for a reference or a scope o, denoted $\mathcal{G} \downarrow o$, if all the resolutions in all its ground instances are the same:

$$\mathcal{G} \downarrow o \triangleq \forall \phi, \phi' \vdash_{\phi(\mathcal{G})} o \longmapsto x_i^{\mathsf{D}} \Rightarrow \vdash_{\phi'(\mathcal{G})} o \longmapsto x_i^{\mathsf{D}}$$

Soundness Given this definition, we can prove that the algorithm on incomplete graphs is correct with respect with the calculus:

Lemma 2. For any incomplete graph $\mathcal{G}$:

$$x_i^{\mathsf{D}} \in R_{\mathcal{G}}(x^{\mathsf{R}}) \implies \vdash_{\mathcal{G}} x^{\mathsf{R}} \longmapsto x_i^{\mathsf{D}} \wedge \mathcal{G} \downarrow x^{\mathsf{R}}$$

where $R_{\mathcal{G}}(x^{\mathsf{R}})$ denotes the top-level resolution function $R[\emptyset](x^{\mathsf{R}})$ for the graph $\mathcal{G}$.

Lemma 1 states that this property holds when the graph $\mathcal{G}$ is ground. We next prove that if the resolution on an incomplete graph $\mathcal{G}$ terminates with a total flag T then for any graph $\mathcal{G}'$ that is an instance of $\mathcal{G}$, the result is the same.

$$Env_{re}[\mathbb{I},\mathbb{S}](x^{\mathsf{R}})_{\mathcal{G}} = (\mathsf{T},\ D) \implies Env_{re}[\mathbb{I},\mathbb{S}](x^{\mathsf{R}})_{\mathcal{G}'} = (\mathsf{T},\ D) \qquad \text{(i)}$$

Proof. We prove this result along with similar result for all the other functions by induction on the termination order of the algorithm. The fact that the result is total implies that the results of all the recursive calls are also total and this allows us to apply the desired induction hypothesis (when a P or U flag is raised it is always propagated). □

Now we show that the resolution is also correct in the partial case. Let $\mathcal{G}$ be an incomplete scope graph and $\mathcal{G}'$ one of its instances. If a resolution on $\mathcal{G}$ contains a set of declarations for a given name then the resolution on $\mathcal{G}'$ contains the same declarations for this name:

$$Env_{re}[\mathbb{I},\mathbb{S}](S)_{\mathcal{G}} = (\_,\ D) \implies Env_{re}[\mathbb{I},\mathbb{S}](S)_{\mathcal{G}'} = (\_,\ D') \implies$$
$$\forall x, \{x^{\mathsf{D}} \in D\} \neq \emptyset \Rightarrow \{x^{\mathsf{D}} \in D\} = \{x^{\mathsf{D}} \in D'\} \quad \text{(ii)}$$

Proof. We prove this result along with similar result for all the other functions by induction on the termination order of the algorithm, using (i). □

Finally, we can prove Lemma 2:

Proof. Let $S_x = R_{\mathcal{G}}(x^{\mathsf{R}})$ and pick $x_i^{\mathsf{D}} \in S_x$. To prove that x^{R} resolves to x_i^{D} in $\mathcal{G}$, let $\mathcal{G}'$ be an arbitrary ground instance of $\mathcal{G}$. Using (ii) we have $x_i^{\mathsf{D}} \in R_{\mathcal{G}'}(x^{\mathsf{R}})$ and by Lemma 1 we have $\vdash_{\mathcal{G}'} x^{\mathsf{R}} \longmapsto x_i^{\mathsf{D}}$. By $\blacklozenge$, we get that $\vdash_{\mathcal{G}} x^{\mathsf{R}} \longmapsto x_i^{\mathsf{D}}$.

To prove stability, let $\mathcal{G}_1$ and $\mathcal{G}_2$ be ground instances of $\mathcal{G}$. Then using (ii), we have $R_{\mathcal{G}_1}(x^{\mathsf{R}}) = R_{\mathcal{G}_2}(x^{\mathsf{R}}) = S_x$, so by definition we have $\mathcal{G} \downarrow x^{\mathsf{R}}$. □

2.4.4 Name Collection Computation

This resolution algorithm on partial graphs is used to compute not only resolution of references but also the set of names visible from a given scope. Given an incomplete graph $\mathcal{G}$ and a scope S, we compute name collections as:

$N_{\mathcal{G}}(\overline{\mathcal{D}}(S)) = \pi(\mathcal{D}_{\mathcal{G}}(S))$ $\qquad$ $N_{\mathcal{G}}(\overline{\mathcal{R}}(S)) = \pi(\mathcal{R}_{\mathcal{G}}(S))$

$N_{\mathcal{G}}(\overline{\mathcal{V}}(S)) = \pi(\{x_i^{\mathsf{D}} \mid \exists E,\ Env_{\mathcal{E}}[\emptyset,\emptyset](S)_{\mathcal{G}} = (\mathsf{T}, E) \wedge x_i^{\mathsf{D}} \in E\})$

(S-Resolve)

$$(x^{\mathsf{R}} \mapsto \delta \wedge C, \mathcal{G}, \psi) \longrightarrow [\delta \mapsto x^{\mathsf{D}}](C, \mathcal{G}, \psi) \qquad \textit{where } x^{\mathsf{D}} \in R_{\mathcal{G}}(x^{\mathsf{R}})$$

(S-Assoc)

$$(x^{\mathsf{D}} \rightsquigarrow \varsigma \wedge C, \mathcal{G}, \psi) \longrightarrow [\varsigma \mapsto S](C, \mathcal{G}, \psi) \qquad \textit{where } x^{\mathsf{D}} \longrightarrow\!\!\triangleright S$$

(S-Equal)

$$(T_1 \equiv T_2 \wedge C, \mathcal{G}, \psi) \longrightarrow \sigma(C, \mathcal{G}, \psi) \qquad \textit{where } \mathcal{U}(T_1, T_2) = \sigma$$

(S-Unique)

$$(!N \wedge C, \mathcal{G}, \psi) \longrightarrow (C, \mathcal{G}, \psi) \qquad \textit{where } \forall x \in N_{\mathcal{G}}(N), \mathbf{1}_{N_{\mathcal{G}}(N)}(x) = 1$$

(S-SubName)

$$(N_1 \lesssim N_2 \wedge C, \mathcal{G}, \psi) \longrightarrow (C, \mathcal{G}, \psi) \qquad \textit{where } N_{\mathcal{G}}(N_1) \subseteq N_{\mathcal{G}}(N_2)$$

(S-TypeOf)

$$(x^{\mathsf{D}} : T \wedge C, \mathcal{G}, \psi) \longrightarrow \begin{cases} (C, \mathcal{G}, \{x^{\mathsf{D}} \mapsto T\} \cup \psi) & \textit{if } x^{\mathsf{D}} \notin \mathrm{dom}(\psi) \\ (\psi(x^{\mathsf{D}}) \equiv T \wedge C, \mathcal{G}, \psi) & \textit{otherwise} \end{cases}$$

(S-True)

$$(\mathsf{True} \wedge C, \mathcal{G}, \psi) \longrightarrow (C, \mathcal{G}, \psi)$$

Figure 2.12: Constraint solving algorithm

Lemma 3 (Name computation soundness). If the computation of a name collection E terminates on an incomplete graph $\mathcal{G}$, its results is the semantics of the name collection for any graph $\mathcal{G}'$ that is an instance of $\mathcal{G}$:

$$N_{\mathcal{G}}(E) = M \Longrightarrow [\![E]\!]_{\mathcal{G}'} = M.$$

2.4.5 Constraint Solving Algorithm

With this name resolution algorithm in hand, Fig. 2.12 gives an algorithm to solve the constraint system from Section 2.3. The algorithm is a non-deterministic rewrite system working over tuples $(C, \mathcal{G}, \psi)$ of a constraint, a scope graph, and a typing environment. It is non-deterministic in the sense that rules may be applied to any atomic constraint in any order considering that $\wedge$ is associative and commutative.

Name resolution introduces ambiguity, since a reference x^{R} may resolve to multiple definitions. If this happens the solver branches, picking a different resolution for x^{R} in every branch. The returned solution is a set of all the $(C, \mathcal{G}, \psi)$ tuples the solver was able to construct. The initial state of the solver is the collected constraint, the (incomplete) scope graph built from the scope graph constraints and an empty typing environment. The algorithm will eliminate clauses from C while instantiating $\mathcal{G}$ and filling ψ. The algorithm terminates when the constraint is empty or no more clauses can be solved. Each rule solves one constraint, possibly updating components of the tuple or applying a substitution to it.

- Rule S-Resolve solves resolution constraints $x^{\mathsf{R}} \mapsto \delta$ using the resolution algorithm from Fig. 2.11. If a resolution is found, it is substituted for the variable δ. If the scope graph is incomplete, the algorithm might return U, in which case the constraint is left to be solved later.

- Rule S-Assoc solves scope association constraints $x^{\mathsf{D}} \rightsquigarrow \varsigma$ by looking up the scope S associated with ground declaration x^{D} in the scope graph. By substituting S for ς, the scope graph becomes more complete, possibly allowing more references to be resolved.

- Rule S-Equal solves equality constraints $T_1 \equiv T_2$. It uses first order unification $\mathcal{U}(T_1, T_2)$, as described by Baader and Nipkow (1998). The resulting substitution is applied to the tuple.

- Rule S-Unique solves $!N$ constraints by checking that the identifier collection N can be computed and all identifiers in it are distinct. ($\mathbf{1}_A(x)$ is the multiplicity of x in A).

- Rule S-SubName solves $N_1 \lesssim N_2$ constraints by checking that the identifier collections N_1 and N_2 can be computed and that every identifier in N_1 is also in N_2.

- Rule S-TypeOf solves type assignment constraints $x^{\mathsf{D}} : T$. The rule considers two cases. When no type assignment is declared for x^{D} in ψ (i.e. the first time that it is encountered) the assignment is added to the typing environment ψ. When a type assignment *is* declared (i.e. for subsequent encounters), the type T from the constraint is unified with the type $\psi(x^{\mathsf{D}})$ from the typing environment.

The constraint resolution algorithm is sound with respect to the constraints semantics.

Lemma 4 (Constraint Solver correctness). If the algorithm produces a solution to a resolution problem then the solution is valid: for all $C, \mathcal{G}, \mathcal{G}', \psi'$:

$$(C, \mathcal{G}, \emptyset) \longrightarrow^{*} (\mathsf{True}, \mathcal{G}', \psi') \implies \exists \phi, \phi(\mathcal{G}) = \mathcal{G}' \wedge \forall \sigma, \sigma\mathcal{G}', \sigma\psi' \models \sigma(\phi(C))$$

Proof. To prove this result we first state some results on the auxiliary unification.

Unification: If $\mathcal{U}(t_1, t_2) = \sigma$ then $\sigma t_1 = \sigma t_2 \wedge \sigma\sigma = \sigma$. See Baader and Nipkow (1998) for a survey on unification problem and unification algorithms for first order terms.

Resolution Soundness: Now we can prove the Lemma 4 of the constraint resolution algorithm. We first prove that for each reduction step, if the output is satisfiable, the input is also satisfiable in the same definition-to-type environment:

$$\begin{aligned}&\forall (C_1, \mathcal{G}_1, \psi_1), (C_2, \mathcal{G}_2, \psi_2), (C_1, \mathcal{G}_1, \psi_1) \longrightarrow (C_2, \mathcal{G}_2, \psi_2) \Longrightarrow \\ &\quad \exists \sigma', \sigma'(\mathcal{G}_1) = \mathcal{G}_2 \,\wedge \left(\forall \sigma, (\sigma(\mathcal{G}_2), \sigma\psi_2) \models \sigma(C_2) \Rightarrow (\sigma\mathcal{G}_2, \sigma\psi_2) \models \sigma\sigma'(C_1)\right) \qquad (2.1)\end{aligned}$$

The proof of this property is by case analysis on the reduction step. From it, we can prove Lemma 4 by a simple induction on the number of reduction steps. □

2.5 Related Work and Discussion

In this section, we discuss the relation of this paper with previous and other related work, and discuss limitations and ideas for future work.

Previous Work The work in this paper is based closely on our previous *theory of name resolution* (Néron et al., 2015), which we extend and generalize here as follows: (i) a scope graph is now defined directly by a set of constraints; (ii) we generalize the parent relation to an arbitrary labeled direct edge between pairs of scopes, and the named import relation to an arbitrary labeled nominal edge between scopes and references; (iii) we extend the resolution algorithm to handle arbitrary well-formedness conditions expressed as regular expressions over arbitrary sets of path labels and arbitrary visibility orderings on labels; (iv) we support partial resolution over incomplete scope graphs; (v) we add the seen-scopes component, previously an artifact of the resolution algorithm, to the resolution calculus to prevent cyclic resolution paths.

The development of the scope graph framework fits in an ongoing line of research to provide high-level domain-specific support for name binding and type analysis in the Spoofax Language Workbench (Kats and Visser, 2010) using the NaBL and TS meta-DSLs (Konat et al., 2012; Visser et al., 2014; Wachsmuth et al., 2013). NaBL is a DSL for defining the name binding rules of programming languages by identifying the references, definitions, scopes, and imports in an abstract syntax tree without recourse to environments or symbol tables (Konat et al., 2012). TS is a complementary DSL for defining type analysis rules. (The design of TS is not formally published, but it is sketched in (Visser et al., 2014).) Rules in TS are similar to traditional typing judgments, relating an expression to a type. However, type rules do not have to propagate context information, since that is taken care of by the separate binding rules. TS rules refer to the results of name analysis produced by NaBL (e.g. `definition of x has type t`), and NaBL rules refer to the results of type analysis to achieve type-dependent name resolution. NaBL and TS are implemented by generation of (1) a language-specific AST traversal that generates 'tasks', and (2) a language-independent task engine that evaluates tasks in order to (incrementally) compute a name and type assignment (Wachsmuth et al., 2013). The resulting name and type analysis engines produce Eclipse IDE support for editor services such as name and type error checking, reference resolution, and code completion.

While NaBL and TS are used in practice to build language definitions with Spoofax, the lack of a solid theoretical foundation was a problem for further development. The aim to verify properties of language definitions (Visser et al., 2014) requires a semantics that can be explained to a proof assistant such as Coq. In particular, the semantics of notions such as imports and 'subsequent scope' were hard to capture. NaBL has some limitations in its coverage of name binding patterns. For example, it cannot express variations on let bindings such as sequential and parallel let. While the task engine is constraint-like, its type resolution is not based on unification, which entails that TS cannot be used to express languages requiring type inference. The constraint language developed in this paper provides a solid formal basis for developing a new generation of name binding and type specification languages.

Prototype Implementation We have developed a prototype implementation of the constraint solver and applied it in the IDE generated with the Spoofax Language Workbench (Kats and Visser, 2010) for the LMR model language used in this paper. However, the prototype does not yet implement the parameterized name resolution algorithm developed in this paper, but uses the fixed policy from Néron et al. (2015). In the prototype implementation, sets of constraints for erroneous programs lead to partial solutions with unsolvable residual constraints that can be translated into error messages in an IDE. However, we have not formalized this; we have only proven the *soundness* of the solver for successful reductions. Furthermore, the implementation is not optimized, nor does it support incremental evaluation of constraints in the sense of the NaBL/TS task engine (Wachsmuth et al., 2013).

Constraints The use of constraints to abstract out type inference problems from the abstract syntax tree is a common approach in implementations and extensions of the Hindley/Milner type system (Milner, 1978) and has been applied to a huge variety of typing features. However, these approaches do not address name resolution using constraints, but rather perform name resolution during constraint collection. For example, in the work of Palsberg and Schwartzbach (1991, 1994) on object-oriented type systems, constraints are associated with identifiers, which requires these to be resolved before constraint collection. We believe that our use of constraints to define static name resolution is novel. Instead of performing name resolution during constraint collection, we provide a reusable set of constraints to express name resolution problems, including name resolution for 'remote' names through imports and the interaction between name and type resolution in type-dependent name resolution.

A variation on traditional type system definitions using inference rules is the cocontextual approach of Erdweg, Bracevac, et al. (2015). Instead of propagating an environment to the sub-terms, environments are 'synthesized' along with type constraints, and the constraints and environments for sub-terms are merged. This allows for compositional and incremental processing of name and type constraints. Name resolution is expressed using operations on environments. It would be interesting to consider a bottom-up collection of constraints in our approach. The extraction algorithm of Fig. 2.6 can be reformulated as a bottom-up collector, using scope variables as placeholders for as yet unknown scopes. However, a key difference with our approach is the support for imports (and nominal instead of structural record types, which requires inspecting the AST associated with a type declaration), which precludes a representation of context information using a flat environment. A general challenge lies in the convergence of these approaches: how to realize incremental name and type analysis in the face of imports?

Attribute Grammars Another common approach to the implementation of static semantic analysis is by means of attribute grammars (Knuth, 1968). In traditional attribute grammars all 'semantic' operations are carried out in the value domain. Thus, name resolution is expressed by propagating a type environment or symbol table through attribute values. Kastens and Waite (1991) provide a reusable ADT for the definition of name analysis that bears some resemblance to our scope graph framework, although the treatment of modules and imports is only discussed at the implementation level. Such attribute grammars would be a suitable mechanism for the definition of constraint collection. The extraction algorithm in Fig. 2.6 could easily be rephrased as an attribute grammar with scopes and type variables as inherited attributes and constraints as synthesized attribute. In *reference attribute grammars* (Hedin, 2000), attributes can get references to tree nodes as values. Thus, attributes can be used to link references (in the scope graph sense) to their declarations. For example, Ekman and Hedin (2006) provide a generic framework for name resolution based on generic reference attributes. Though this framework is part of the JastAdd Java compiler, it can be reused for other languages as well. The framework needs to be instantiated with language-specific lookup functions to resolve names. These can be specified modularly per language construct, making it possible to echo the structure of the Java language specification of name binding closely. However, these lookup functions programatically encode name binding idioms such as lexical scoping, shadowing, and hiding. Reference attributes can also be used in the specification of type analysis. Similar to our approach, name binding and typing rules can be specified mostly separately. In a generic framework, Ekman and Hedin (2007a) use reference attributes to link language constructs to their types and to represent type relations such as subtyping. Similar to name resolution, instantiations of the framework need to be encoded programatically. Modularity and extensibility require particular encoding patterns such as double dispatch.

The distinctive feature of our approach is that we treat name resolution using a largely separate mechanism, the scope graph, rather than integrating it into type resolution. Since some language constructs require type-dependent name resolution, there is inevitably some interaction between naming and typing, but we are still able to reuse most of our existing name resolution theory, which gives us the ability to handle a very rich variety of name binding schemes.

Future Work There are many directions for future work. One important goal is to extend our theory to handle languages with more sophisticated typing features, including subtyping, type-parameterized classes and functions, and modules with type signatures. To support popular OO language idioms, we also need to add support for multiple independent name spaces (and disambiguation across them) and type-based overloading resolution. As we make such extensions, we would also like to address the completeness of the constraint resolution algorithm (on suitably restricted sets of constraints). In particular, it would be interesting to integrate approaches to type error recovery (Heeren et al., 2003; D. Zhang and Myers, 2014; D. Zhang et al., 2015) in order to generate good quality type error messages automatically.

On a pragmatic front, more analysis and implementation experiments are needed to determine if our approach will scale to real-world tools. In particular, we need to assess the theoretical and actual efficiency of our constraint solving algorithm. In addition, many applications for semantic analysis (e.g. in IDEs) require efficient incremental computation of name and type resolution.

On the usability front, we are interested in evaluating the expressivity and understandability of our constraint language and of higher-level name and type specification languages that we express in terms of it. Is there a payoff to the use of high-level, but perhaps more abstract concepts, in contrast to a direct implementation?

Finally, we are interested in extending the application of our building block approach to other tasks where constraint-based methods have proved useful, such as pointer analysis.

Acknowledgments We thank the anonymous reviewers for their feedback on previous versions of this paper. 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.

Scopes as Types

3

Abstract 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.

3.1 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.

Modeling Names in Programming Languages Formal definitions of type systems and their implementation as type checkers feature a variety of techniques to model and implement name binding and name resolution for different languages. For example, if we consider Pierce's (2002) book we encounter the following representations for

Published as Hendrik van Antwerpen, Casper Bach Poulsen, Arjen Rouvoet, and Eelco Visser (2018). "Scopes as types." In: *Proceedings of the ACM on Programming Languages* OOPSLA. DOI: 10.1145/3276484. Copyright © 2018 Owner/Author. Licensed under a Creative Commons Attribution International 4.0 License.

the treatment of names: sequences of name-type associations to represent type environments for the simply-typed lambda calculus; tuples of label-type associations to represent record and variant types; class tables with functions for field and method lookup to represent the nominal class types of Featherweight Java; types with quantifiers to represent parameterized types in System F; and pairs of type variables and types to represent existential types. These are all fine mathematical representations, but they have been optimized for the particular language they model. These optimizations obscure the understanding of the common underlying concepts of name binding. Furthermore, the variation in representations is not a good basis for the construction of reusable tools for language design. Would it be possible to standardize the treatment of names in programming languages?

Modeling Name Resolution with Scope Graphs Scope graphs were introduced by Néron et al. (2015) as a general model for name resolution in programming languages that is suitable for formalization as well as implementation. A scope graph captures the binding structure of a program. A scope is a region in a program that behaves uniformly with respect to name resolution. Declarations of names and references are associated with scopes. Visibility is modeled by edges between scopes. A generic, language-independent resolution algorithm interprets a scope graph to resolve references to declarations by finding the most specific well-formed path in a scope graph. To express the binding rules of a programming language, one defines a mapping from abstract syntax trees to scope graphs. Scope graphs cover a wide range of binding structures, including *lexical bindings*[1] such as let bindings, function parameters, and local variables in blocks; and *non-lexical bindings* such as (potentially cyclic) module imports and class inheritance. The framework enables language-independent definitions of alpha equivalence and safe variable renaming.

The scope graph framework has already been used succesfully in several applications. van Antwerpen et al. (2016) use scope graphs to model name binding in a constraint language for the definition of type checkers. Bach Poulsen et al. (2016) define a framework in which *scopes describe frames*, providing a language-independent model for run-time memory in dynamic semantics. Bach Poulsen et al. (2018) show that this model can be used to realize *type safety by construction* in intrinsically-typed definitional interpreters for imperative languages.

Thus, scope graphs are the basis for a promising approach to the definition of the static semantics of programming languages that serves the implementation of tools such as type checkers, as well as the verification of language properties such as type safety. However, the adoption of scope graphs is inhibited by its limitation to simple type systems. As a model that ties information to *names*, scope graphs

[1] Lexical bindings are those in which the name binding construct dominates the abstract tree that corresponds to the scope of the construct. Non-lexical bindings define names that are reachable outside the dominated tree.

appear to be limited in expressiveness. The works cited above cover languages with *simple, nominal type systems* in which types are identified *by name*, and their future work calls for extension to more sophisticated type systems. In particular, it is not clear how scope graphs can be used to describe *structural types*, in which types are not identified by name, and *generic types*, in which types are parameterized by types.

Scopes as Types In this paper, we demonstrate how scope graphs *can* be used to model type systems with more sophisticated forms of type representation and compatibility checking, such as structural record types and parameterized types in both nominal and structural type systems, by using *scopes as types*. Scope graph scopes can model a variety of structured types such as records and classes. Visibility edges between scopes can be used to model subtyping. The instantiation of a parameterized type can be modeled by means of a new scope that refines the binding of a parameter. To realize scopes-as-types we generalize scope graphs with scoped relations, formalizing scoped information including typed declarations, and we simplify scope graphs by using the scopes-as-types approach to model imports, which were previously built into the framework.

We demonstrate how the approach can be applied in the definition of type systems for the simply-typed lambda calculus with records (featuring structural subtyping; Pierce, 2002), System F (featuring parametric types; Girard, 1972; Reynolds, 1974), and Featherweight Generic Java (featuring generic class types; Igarashi et al., 2001).

Staging Name Resolution and Type Checking Scope graphs provide a uniform model for the representation and resolution of binding information in programs, but they do not, by themselves, address another issue with realizing declarative definition of type checkers: the staging of name resolution and type checking. It is common practice to use *constraints* in type checkers in order to separate the collection of type compatibility requirements, and checking that these are satisfied. However, name resolution is typically performed during the traversal of the abstract syntax tree that generates constraints. This requires a careful staging of the traversal in order to collect information (names and their types) before it will be needed. For example, checking a recursive let expression requires processing the declared variables before checking the initializing expressions. Similarly, checking modules or classes requires collecting signature information before checking their contents. This approach is further complicated when considering *type-dependent name resolution* in which the resolution of names depends on the resolution of types.

In this paper, we introduce *Statix*, a constraint-based declarative language for the specification of type systems that combines type constraints with *name resolution constraints* based on scope graphs. That is, Statix rules define the static semantics of language constructs in terms of constraints over type terms and constraints that define and query a scope graph. Definition of type checkers using this approach is more declarative since the order of evaluation of constraints is not tied to the order of the traversal of the abstract syntax tree. In particular, this relieves the language designer from ensuring that information is collected before it is used. Statix generalizes the constraint language of van Antwerpen et al. (2016) by introducing user-defined constraints, required to define type compatibility predicates, and by generalizing name resolution to scope graph queries to retrieve (visible) scoped information.

Contributions The paper makes the following technical contributions:

- We show that viewing *scopes as types* enables modeling the internal structure of types in a range of interesting type systems, including structural records and generic classes, using the generic representation of scopes.
- We extend the scope graph framework of Néron et al. (2015) and van Antwerpen et al. (2016) with *scoped relations* to model the association of types with declarations and the representation of explicit substitutions in the instantiation of parameterized types. We generalize name resolution from resolution of references to general *queries* for scoped relations. Furthermore, visibility policies, which were global (per language), can be defined per query, enabling namespace-specific visibility policies. We *simplify* the framework by not including imports as a primitive, since these can be encoded using the scopes-as-types approach.
- We extend the *visual notation* of scope graph diagrams with scoped relations, which provides a useful language to explain patterns of names and types in programming languages.
- We introduce *Statix*, a declarative language to specify type systems. The language provides simple guarded rules for the definition of user-defined constraints with unification, scope graph construction, and name resolution as built-in theories. We provide a formal definition of the declarative semantics of Statix.
- We discuss the execution model of Statix and how it guarantees soundness of *resolution in incomplete graphs*.
- We evaluate the scopes-as-types approach and the Statix language in three case studies: the simply-typed lambda calculus with records (STLC-REC, featuring structural sub-typing; Pierce, 2002), System F (featuring parametric types; Girard, 1972; Reynolds, 1974), and Featherweight Generic Java (featuring generic class types; Igarashi et al., 2001).

Outline In Section 3.2 we present the revised scope graph framework and the corresponding resolution calculus. We demonstrate how this formalism supports the specification of type systems, including ones with structural and parametric types. In

Section 3.3 we introduce the Statix language and its declarative semantics. We show the specification in Statix of typical patterns in programming languages with structural and parametric type systems. In Section 3.4 we discuss the execution model of the solver for the Statix language. In particular, we discuss resolution in incomplete scope graphs. In Section 3.5 we discuss the evaluation of Statix by means of an implementation in the Spoofax language workbench and several critical case studies. In Section 3.6 we discuss how the approach compares to other approaches. We conclude in Section 3.7.

3.2 Scopes as Types

Typing is deeply dependent on name resolution: a program phrase is typically typed by resolving names that occur in it to names in its surrounding context. In many interesting languages, types can also bind names; this is the case with record types, object or class types, and dependent types. In this section we observe and illustrate how types that bind names (records, objects, etc.) can be described by scopes in a scope graph, and we present a revised definition of the scope graph framework of Néron et al. (2015) and van Antwerpen et al. (2016) and show how it can be applied to the definition of type systems.

3.2.1 Scope Graphs and the Resolution Calculus

In the scope graph approach, a program is reduced to a graph that represents its binding information. The first part of Fig. 3.1 defines the structure of scope graphs. A scope graph consists of *scopes*, connected by *edges*, containing *data*. A *labeled edge* $s_1 \xrightarrow{l} s_2$ between scopes s_1 and s_2 determines that the declarations in scope s_2 are reachable from scope s_1. The label can be used to regulate visibility. A *scoped datum* $s \overset{r}{\relbar\!\!\!\blacksquare}\ d$ associates a data term d with a scope s under relation r. For example, we will use $s \overset{:}{\relbar\!\!\!\blacksquare}\ (x, T)$, to represent a declaration of name x in the scope s with type T, and use $x : T$ to denote the pair. There may be multiple data items associated with a scope under the same relation.

Given this structure, we can now precisely characterize *name resolution* for a *reference* as finding a *path* from its scope to a scope with a matching *declaration*. This intuition is formally captured by the *resolution calculus* in the third part of Fig. 3.1, which is parameterized by well-formedness and visibility parameters defined in the second part of Fig. 3.1. We discuss the judgments of the calculus.

The judgment $\mathcal{G} \vdash p : s_1 \twoheadrightarrow s_2$ states that there is a *path* p from scope s_1 to scope s_2, if there is a sequence of labeled scope edges starting at s_1 and leading to s_2. Cyclic paths are not admitted: the $s_1 \notin \mathsf{scopes}(p)$ premise of (NR-Cons) asserts that scope s_1 does not occur in path p. The path p records the scopes and edge labels that it passes through.

Syntax Parameters

data terms	$d \in \mathcal{D}$	a set of data terms
labels	$l \in \mathcal{L}$	a set of edge labels
relations	$r \in \mathcal{R}$	a set of relation names

Syntax Definitions

scopes	$s \in \mathcal{S}$	$:=$	some countable set
paths	$p \in \mathcal{P}$	$::=$	$s \mid s{\cdot}l{\cdot}p$
edges	Edges	$::=$	$s \xrightarrow{l} s$
datums	Data	$::=$	$s \overset{r}{\relbar\!\!\blacksquare}\ d$
scope graphs	$\mathcal{G} \in \mathit{Graphs}$	$::=$	$\langle \mathsf{scopes} \subseteq \mathcal{S}, \mathsf{edges} \subseteq \mathit{Edges}, \mathsf{data} \subseteq \mathit{Data} \rangle$
extended labels	$\hat{l} \in \hat{\mathcal{L}}$	$:=$	$\mathcal{L} \cup \{\$\}$ where \$ indicates the end of a path

Visibility Parameters

data term well-formedness	$\mathit{WFD} \subseteq \mathcal{D}$	
label well-formedness	$\mathit{WFL} \subseteq \mathcal{L}^*$	defined as a regular expression
data order	$\leq_d \ \subseteq \mathcal{D} \times \mathcal{D}$	partial order
label order	$<_l \ \subseteq \hat{\mathcal{L}} \times \hat{\mathcal{L}}$	strict partial order

Path Well-formedness $\boxed{\mathit{WFL} \vdash p \text{ OK}}$

$$\frac{(l_1 \dots l_n) \in \mathit{WFL}}{\mathit{WFL} \vdash (s_1{\cdot}l_1{\cdot}\dots{\cdot}s_n{\cdot}l_n{\cdot}s_{n+1}) \text{ OK}}$$

Visibility Order $\boxed{<_l \vdash p <_p p}$

$$\frac{<_l \vdash p_1 <_p p_2}{<_l \vdash s{\cdot}l{\cdot}p_1 <_p s{\cdot}l{\cdot}p_2} \qquad \frac{\$ <_l l}{<_l \vdash s <_p s{\cdot}l{\cdot}p} \qquad \frac{l <_l \$}{<_l \vdash s{\cdot}l{\cdot}p <_p s} \qquad \frac{l_1 <_l l_2}{<_l \vdash s{\cdot}l_1{\cdot}p_1 <_p s{\cdot}l_2{\cdot}p_2}$$

Paths $\boxed{\mathcal{G} \vdash p : s \twoheadrightarrow s}$

$$\text{(NR-Refl)}\ \frac{s \in \mathsf{scopes}(\mathcal{G})}{\mathcal{G} \vdash s : s \twoheadrightarrow s} \qquad \text{(NR-Cons)}\ \frac{s_1 \xrightarrow{l} s_2 \in \mathsf{edges}(\mathcal{G}) \quad \mathcal{G} \vdash p : s_2 \twoheadrightarrow s_3 \quad s_1 \notin \mathsf{scopes}(p)}{\mathcal{G} \vdash s_1{\cdot}l{\cdot}p : s_1 \twoheadrightarrow s_3}$$

Reachability $\boxed{\mathit{WFD}, \mathit{WFL}, \mathcal{G} \vdash p : s \overset{r}{\rightarrowtail} d}$

$$\text{(NR-Rel)}\ \frac{\mathcal{G} \vdash p : s \twoheadrightarrow s' \quad s' \overset{r}{\relbar\!\!\blacksquare}\ d \in \mathsf{data}(\mathcal{G}) \quad \mathit{WFL} \vdash p \text{ OK} \quad d \in \mathit{WFD}}{\mathit{WFD}, \mathit{WFL}, \mathcal{G} \vdash p : s \overset{r}{\rightarrowtail} d}$$

Visibility $\boxed{\mathit{WFD}, \mathit{WFL}, \leq_d, <_l, \mathcal{G} \vdash p : s \overset{r}{\mapsto} d}$

$$\text{(NR-Vis)}\ \frac{\begin{array}{c}\mathit{WFD}, \mathit{WFL}, \mathcal{G} \vdash p : s \overset{r}{\rightarrowtail} d \\ \nexists\, p'd'.\, \Big(\big(\mathit{WFD}, \mathit{WFL}, \mathcal{G} \vdash p' : s \overset{r}{\rightarrowtail} d'\big) \wedge (<_l \vdash p' <_p p) \wedge (d' \leq_d d)\Big)\end{array}}{\mathit{WFD}, \mathit{WFL}, \leq_d, <_l, \mathcal{G} \vdash p : s \overset{r}{\mapsto} d}$$

Figure 3.1: Formal definition of scope graphs with syntax, visibility predicates, and resolution calculus

The judgment $\mathit{WFD}, \mathit{WFL}, \mathcal{G} \vdash p : s \overset{r}{\rightarrowtail} d$ states that data term (declaration) d is *reachable* through path p from scope s under relation r with data term predicate *WFD* and label well-formedness predicate *WFL*. Label well-formedness tests that the path has a 'good shape' as defined by a regular expression. This is used to model policies such as transitive vs. non-transitive imports or the unreachability of lexical parents of imported modules (van Antwerpen et al., 2016). Data term well-formedness tests whether we have found the datum we were looking for. For example, to resolve a reference x we use a well-formedness predicate that matches all declaration-type pairs $y : T$ where $x \simeq y$, that is, the reference has the same name as the declaration (but a different position in the program).

Finally, the judgment $\mathit{WFD}, \mathit{WFL}, \leq_d, <_l, \mathcal{G} \vdash p : s \overset{r}{\longmapsto} d$ states that data term d is *visible* through path p from scope s under relation r with the well-formedness predicates *WFD*, *WFL*, and the orders $\leq_d$ and $<_l$. The parameterization is chosen such that algorithmic resolution remains feasible (see Section 3.4.2). The *visibility order* $p_1 <_p p_2$ ('p_1 shadows p_2') is defined as a prefix order over the labels of a path, in terms of a label order $<_l$. A special label \$ indicates the end of a path, and is used to order paths of different lengths. The prefix order only orders paths that have a common prefix. That is, $s_1{\cdot}l_1{\cdot}s_2 \not<_p s'_1{\cdot}l'_1{\cdot}s'_2{\cdot}l'_2{\cdot}s'_3$ when $s_1 \neq s'_1$ or $l_1 \neq l'_1$. The *data order* $d \leq_d d$ decides which declarations shadow each other, if multiple declarations are reachable via shadowing paths. This is used to specify all visible declarations, where a declaration only shadows a declaration that is reached via a shadowed path, if it has the same name. Using the visibility order and data order, the rule (NR-Vis) defines that a data term d is visible through path p when there does not exist a data term d' reachable through p' such that d' shadows d and such that p' is strictly preferred over p. We will illustrate below how this captures the notion of shadowing in name resolution.

In the rest of this section we show how scope graphs can be used in the definition of type systems for languages with a variety of binding systems, including bindings in types. We discuss how our approach compares to representations of binding in traditional definitions of type systems.

3.2.2 Simply-Typed Lambda Calculus

First we consider the syntax and typing rules for the simply-typed lambda calculus with arithmetic expressions (STLC) in Fig. 3.3. The language consists of number constants, addition, function literals, variables, function application, and let bindings.

Name binding for STLC is typically modeled using type environments, which are ordered lists of pairs associating a name with a type. Scoping is modeled by extension of an environment with a new pair, which shadows any earlier declarations of the same name (either by removing a matching pair or through definition of the lookup function). The extended environment is only used for those sub-expressions where the binding is in scope. Scope graphs make the shadowing rules explicit by separating the construction of the binding structure and the definition of resolution.

The typing rules in Fig. 3.3 use scope graphs and the resolution calculus instead of type environments to model binding in STLC. The judgment $\mathcal{G}, s \vdash \texttt{e} : \texttt{t}$ states that in the context of scope graph $\mathcal{G}$ and scope s, expression $\texttt{e}$ has type $\texttt{t}$. The rules are implicitly parameterized by a scope graph $\mathcal{G}$, and use $s_1 \xrightarrow{l} s_2$ as a shorthand for $s_1 \xrightarrow{l} s_2 \in \mathsf{edges}(\mathcal{G})$, and $s \xrightarrow{r}\blacksquare\, d$ for $s \xrightarrow{r}\blacksquare\, d \in \mathsf{data}(\mathcal{G})$. The notation ∇s is used to assert that a scope s is distinct in a scope graph.[2] For example, the ∇s_2 premise in the (STLC-Fun) rule asserts that the scope s_2 is distinct from s_1 in the scope graph.

$\texttt{let } \texttt{x}_1 \texttt{ = 3 in}$
$\texttt{let } \texttt{f}_2 \texttt{ = fun(}\texttt{x}_3 \texttt{ : num) \{ } \texttt{x}_4 \texttt{ \} in}$
$\texttt{f}_5\ \texttt{x}_6$

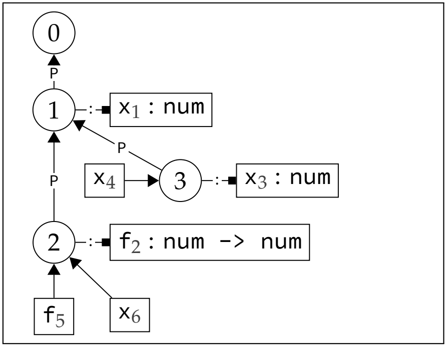


Figure 3.2: A program with nested lets

Scope Graph Structure In addition to defining the *types* of expression forms, the typing rules define the scoping structure of expressions by relating the scope of an expression to the scope(s) of its sub-expressions. Numbers, addition, and function application are non-binding, non-scoping constructs. Thus, rules (STLC-Num), (STLC-Plus), and (STLC-App) state that the scope of the sub-expressions (if any) of these operators is the same as the scope of the parent. Rules (STLC-Fun) and (STLC-Let) introduce a distinct scope s_2 and associate a declaration $\texttt{x}_i : \texttt{t}$ for the binding occurrence with that scope. A scope edge $s_2 \xrightarrow{\mathsf{P}} s_1$ makes the declarations reachable from s_1, also reachable from the scope s_2, which is used as the scope for the sub-expression in which the binding occurrence is in scope: the bodies of the function and let expression. Note that in (STLC-Let), scope s_1 is used for the initialization expression, reflecting that the variable introduced is not in scope in that expression.

Before we consider the rule (SLTC-Id) and name resolution, it can be helpful to visualize scope graphs using *scope graph diagrams*. To distinguish different occurrences of the same name in a program we subscript names in programs by a position index. For example, the program `fun(x : num){ x }` is written $\texttt{fun(}\texttt{x}_1 \texttt{ : num)\{ } \texttt{x}_2 \texttt{ \}}$. Fig. 3.2 shows an example program and its scope graph diagram. Scopes are depicted by circles labeled with a number, and edges between scopes are depicted as labeled edges $\xrightarrow{l}$. Scope #0 in the scope graph in Fig. 3.2 is the scope of the context of the outer let. Scopes #1 and #2 are the scopes of the first and second let, respectively.

[2] We can think of ∇s as a claim to "ownership" of scope s. Each scope in the scope graph can have exactly one "owner". In Section 3.4 we give a declarative semantics of Statix where this notion is formally defined.

Syntax

$$\begin{array}{rlll}
\text{integers} & z \in \mathbb{Z} & := & \{\ldots,\texttt{-1},\texttt{0},\texttt{1},\ldots\} \\
\text{identifiers} & \texttt{x} \in \mathit{Id} & := & \text{some countable set} \\
\text{expressions} & \texttt{e} \in \mathit{Expr} & ::= & z \mid \texttt{e + e} \mid \texttt{fun(x : t)\{ e \}} \mid \texttt{x} \mid \texttt{e e} \mid \texttt{let x = e in e} \\
\text{types} & \texttt{t} \in \mathit{Type} & ::= & \texttt{num} \mid \texttt{t -> t}
\end{array}$$

Resolution Syntax Parameters

$$\begin{array}{rlll}
\text{labels} & l \in \mathcal{L} & ::= & \texttt{P} \\
\text{relations} & r \in \mathcal{R} & ::= & : \\
\text{data terms} & d \in \mathcal{D} & ::= & \texttt{x}_i : \texttt{t}
\end{array}$$

Matching Declarations, Label Order and Data Order $\boxed{d \in \textsc{decl}(\texttt{x}_i)}$ $\boxed{\hat{l} <_l \hat{l}}$ $\boxed{t \leq_{\top} t}$

$$\frac{\texttt{x}_i \simeq \texttt{x}_j}{(\texttt{x}_j : \texttt{t}) \in \textsc{decl}(\texttt{x}_i)} \qquad \$ <_l P \qquad t_1 \leq_{\top} t_2$$

Typing Rules $\boxed{\mathcal{G}, s \vdash \texttt{e} : \texttt{t}}$

$$\text{(STLC-Num)}\frac{}{s \vdash z : \texttt{num}} \qquad \text{(STLC-Plus)}\frac{s \vdash \texttt{e}_1 : \texttt{num} \quad s \vdash \texttt{e}_2 : \texttt{num}}{s \vdash \texttt{e}_1 \texttt{ + } \texttt{e}_2 : \texttt{num}}$$

$$\text{(STLC-Fun)}\frac{\nabla s_2 \quad s_2 \xrightarrow{\texttt{P}} s_1 \quad s_2 \xrightarrow{:} \texttt{x}_i : \texttt{t}_1 \quad s_2 \vdash \texttt{e} : \texttt{t}_2}{s_1 \vdash \texttt{fun(x}_i \texttt{ : t}_1\texttt{)\{ e \}} : \texttt{t}_1 \texttt{ -> } \texttt{t}_2}$$

$$\text{(STLC-Id)}\frac{\textsc{decl}(\texttt{x}_i), \texttt{P}^*, \leq_{\top}, <_l \vdash p : s \mapsto \texttt{x}_j : \texttt{t}}{s \vdash \texttt{x}_i : \texttt{t}} \qquad \text{(STLC-App)}\frac{s \vdash \texttt{e}_1 : \texttt{t}_1 \texttt{ -> } \texttt{t}_2 \quad s \vdash \texttt{e}_2 : \texttt{t}_1}{s \vdash \texttt{e}_1 \ \texttt{e}_2 : \texttt{t}_2}$$

$$\text{(STLC-Let)}\frac{s_1 \vdash \texttt{e}_1 : \texttt{t}_1 \quad \nabla s_2 \quad s_2 \xrightarrow{\texttt{P}} s_1 \quad s_2 \xrightarrow{:} \texttt{x}_i : \texttt{t}_1 \quad s_2 \vdash \texttt{e}_2 : \texttt{t}_2}{s_1 \vdash \texttt{let x}_i \texttt{ = e}_1 \texttt{ in e}_2 : \texttt{t}_2}$$

Figure 3.3: Syntax and typing rules for a simply-typed lambda calculus using scope graphs

Scope #3 is the scope of the function literal. The scopes are connected via P-labeled edges to their lexical parent scope (thus P is for *parent*). Declarations are depicted as boxes associated with scopes via an $\xrightarrow{:}$ edge going *from* a scope *to* a declaration. Lastly, references are depicted as boxes connected to scopes by edges going *from* the reference *to* the scope. References are not formally part of the structure of scope graphs (Fig. 3.1), but we include them in scope graph diagrams to indicate which scope each reference is resolved relative to.

Reachability and Visibility Now we can consider how variables in STLC are resolved in rule (STLC-Id). The premise of the rule states that $\texttt{x}_i$ has type t if $\texttt{x}_i$ can be resolved in scope s through path p leading to a declaration $\texttt{x}_j$: t, such that there is no other declaration with a matching name that shadows $\texttt{x}_j$: t. The data well-formedness and label order used as parameters to resolve $\texttt{x}_i$ in (STLC-Id) are defined in Fig. 3.3. The declaration well-formedness predicate $\textsc{decl}(\texttt{x}_i)$ is parameterized by an identifier $\texttt{x}_i$ with position subscript i as input, and identifies the set of all declarations with the same name as, but at positions different from, $\mathtt{x}_i$. The label well-formedness predicate for STLC is $\mathtt{P}^*$, which reflects that a variable can be resolved in any scope reachable through a sequence of parent edges. Declarations are not ordered in STLC, and the $\leq_\top$ order passed as parameter to the visibility judgment in (STLC-Id) is the order where all declarations are equal. The definition $\mathtt{\$} <_l \mathtt{P}$ of the visibility ordering for STLC specifies that shorter paths are preferred over longer paths. Thus, declaration in a scope that has a path with fewer $\mathtt{P}$ edges is preferred, which formalizes the usual notion of shadowing based on on lexical proximity. For example, the reference $\mathtt{x}_4$ in the program in Fig. 3.2 reaches $\mathtt{x}_1$ and $\mathtt{x}_3$ since $\textsc{decl}(\mathtt{x}_4), \mathtt{P}^* \vdash s_3 \cdot \mathtt{P} \cdot s_2 \cdot \mathtt{P} \cdot s_1 : s_3 \overset{\cdot}{\rightarrowtail} \mathtt{x}_1 : \mathbf{num}$ *and* $\textsc{decl}(\mathtt{x}_4), \mathtt{P}^* \vdash s_3 : s_3 \overset{\cdot}{\rightarrowtail} \mathtt{x}_3 : \mathbf{num}$. However, because $s_3 <_p s_3 \cdot \mathtt{P} \cdot s_2$, we have that $\mathtt{x}_4$ resolves to $\mathtt{x}_3$.

3.2.3 Records and Structural Subtyping

Next we consider an extension of STLC with structural records. The language defined in Fig. 3.4 extends STLC with record literals, field access, record extension, and a Pascal/JavaScript-like **`with`** expression. Record types are *structural*, that is, record types are not identified by name, but by a set of field-type pairs. The type system features *subtyping* between record types: a function expecting a record as parameter can be provided any extension of the expected record type. We discuss how to identify, represent, compose, access, and compare record types.

Identifying Record Types In *nominal* type systems, types are identified *by name*. Information about the type is associated with that name. For example, with scope graphs we can state $s \xrightarrow{td}\blacksquare\, (\mathtt{Point}, r_{\mathtt{Point}})$, which associates with the type name `Point` some representation $r_{\mathtt{Point}}$ of the record type. A record type can then be represented as $\mathsf{REC}(\mathtt{Point})$ referring to the declaration of the type by its name. Such a representation is efficient since copying the type entails copying *a reference* to its representation. Furthermore, a type is directly related to its origin in a program. The disadvantage of nominal types is that each variation of a type must be given a name and that comparisons must be organized through relations between names. In *structural* type systems, types are identified by their structure (Cardelli, 1988). This means that new types can be created 'on the fly', that is, not all types have to be defined by name. In previous work, van Antwerpen et al. (2016) show how to represent nominal record types with scope graphs, but not how to express structural comparisons and composition of such types. Here we show how to do that using scopes as types and scope graph queries.

Representing Record Types The representation of record types requires a mapping from field names to types. Pierce (2002) uses association lists to represent record types. With scope graphs, we do not need a new representation: scopes provide a

Syntax

$$\begin{array}{llll}
\text{expressions} & \mathtt{e} \in \mathit{Expr} & ::= & \ldots \mid \{(\mathtt{x}\ \mathtt{=}\ \mathtt{e})^*\} \mid \mathtt{e.x} \mid \mathtt{e}\ \mathtt{\textbf{extends}}\ \mathtt{e} \mid \mathtt{\textbf{with}}\ \mathtt{e}\ \mathtt{\textbf{do}}\ \mathtt{e} \\
\text{syntactic types} & \mathtt{t} \in \mathit{TypeExpr} & ::= & \mathtt{\textbf{num}} \mid \mathtt{t}\ \mathtt{->}\ \mathtt{t} \mid \{\ (\mathtt{x}\ :\ \mathtt{t})^*\} \\
\text{semantic types} & T, U \in \mathit{Type} & ::= & \mathsf{NUM} \mid T \to T \mid \mathsf{REC}(s)
\end{array}$$

Resolution Syntax Parameters

labels $l \in \mathcal{L} ::= \ldots \mid \mathsf{R} \mid \mathsf{E}$ and otherwise like STLC

Syntactic to Semantic Typing $\boxed{\mathcal{G} \vdash [\![\mathtt{t}]\!] \Rrightarrow T}$

$$\text{(T-Num)}\frac{}{\vdash [\![\mathtt{\textbf{num}}]\!] \Rrightarrow \mathsf{NUM}} \qquad \text{(T-Fun)}\frac{\vdash [\![\mathtt{t}_1]\!] \Rrightarrow T_1 \quad \vdash [\![\mathtt{t}_2]\!] \Rrightarrow T_2}{\vdash [\![\mathtt{t}_1\ \mathtt{->}\ \mathtt{t}_2]\!] \Rrightarrow T_1 \to T_2}$$

$$\text{(T-Rec)}\frac{\vdash [\![\bar{\mathtt{t}}]\!] \Rrightarrow \bar{T} \quad \nabla s_r \quad s_r \overset{:}{\longrightarrow\!\blacksquare} \bar{\mathtt{x}}_i : \bar{T}}{\vdash [\![\{\ \bar{\mathtt{x}}_i\ :\ \bar{\mathtt{t}}\ \}]\!] \Rrightarrow \mathsf{REC}(s_r)}$$

Typing Rules (Records) $\boxed{\mathcal{G}, s \vdash \mathtt{e} : T}$

$$\text{(ERS-Rec)}\frac{s \vdash \bar{\mathtt{e}} : \bar{T} \quad \nabla s_r \quad s_r \overset{:}{\longrightarrow\!\blacksquare} \bar{\mathtt{x}}_i : \bar{T}}{s \vdash \{\ \bar{\mathtt{x}}_i\ \mathtt{=}\ \bar{\mathtt{e}}\ \} : \mathsf{REC}(s_r)}$$

$$\text{(ERS-Access)}\frac{s \vdash \mathtt{e} : \mathsf{REC}(s_r) \quad \textsc{decl}(\mathtt{x}_i), (\mathsf{R}|\mathsf{E})^*, \leq_\top, <_l \vdash p : s_r \mapsto \mathtt{x}_j : T}{s \vdash \mathtt{e.x}_i : T}$$

$$\text{(ERS-Extends)}\frac{s \vdash \mathtt{e}_1 : \mathsf{REC}(s_1) \quad s \vdash \mathtt{e}_2 : \mathsf{REC}(s_2) \quad \nabla s_r \quad s_r \xrightarrow{\mathsf{R}} s_1 \quad s_r \xrightarrow{\mathsf{E}} s_2}{s \vdash \mathtt{e}_1\ \mathtt{\textbf{extends}}\ \mathtt{e}_2 : \mathsf{REC}(s_r)}$$

$$\text{(ERS-With)}\frac{s \vdash \mathtt{e}_1 : \mathsf{REC}(s_r) \quad \nabla s_w \quad s_w \xrightarrow{\mathsf{R}} s_r \quad s_w \xrightarrow{\mathsf{P}} s \quad s_w \vdash \mathtt{e}_2 : T}{s \vdash \mathtt{\textbf{with}}\ \mathtt{e}_1\ \mathtt{\textbf{do}}\ \mathtt{e}_2 : T}$$

Label Order $\boxed{\hat{l} <_l \hat{l}}$

$$\$ <_l \mathsf{P} \qquad \$ <_l \mathsf{R} \qquad \$ <_l \mathsf{E} \qquad \mathsf{R} <_l \mathsf{P} \qquad \mathsf{R} <_l \mathsf{E}$$

Subtyping $\boxed{\mathcal{G} \vdash T <: T}$

$$\text{(<:-Num)}\frac{}{\vdash \mathsf{NUM} <: \mathsf{NUM}} \qquad \text{(<:-Fun)}\frac{\vdash T_2 <: T_1 \quad \vdash U_1 <: U_2}{\vdash T_1 \to U_1 <: T_2 \to U_2}$$

$$\text{(<:-Rec)}\frac{\begin{array}{c}\forall \mathtt{x}_i\ p\ \mathtt{x}_j\ T.\textsc{decl}(\mathtt{x}_i), (\mathsf{R}|\mathsf{E})^*, \leq_\top, <_l \vdash p : s_2 \mapsto \mathtt{x}_j : T \implies \\ \exists p'\ U\ \mathtt{x}_k.\textsc{decl}(\mathtt{x}_i), (\mathsf{R}|\mathsf{E})^*, \leq_\top, <_l \vdash p' : s_1 \mapsto \mathtt{x}_k : U \wedge \vdash U <: T\end{array}}{\vdash \mathsf{REC}(s_1) <: \mathsf{REC}(s_2)}$$

Figure 3.4: Syntax and typing rules for a language with extensible records. The expression syntax is extended from Fig. 3.3. The typing rules for functions are mostly the same as in Fig. 3.3

natural representation for record types. For example, the `x` coordinate of a `Point` type is represented as a declaration in the scope: $s_{\mathtt{Point}} \xrightarrow{:} \mathtt{x} : \mathtt{num}$. Such a scope could be associated with a type name to realize a nominal type system, as discussed above. To realize a structural type system, *we use the scope reference itself as a type*, and represent a record type as $\mathsf{REC}(s_r)$. A difference with the traditional representation of structural types as association lists is that scopes *have identity*. Thus, copying types entails copying of references.

Since scopes are not part of the surface syntax of types, Fig. 3.4 defines two notions of types: *syntactic types* and *semantic types* for use in typing rules. Fig. 3.4 defines a relation $\vdash [\![\mathtt{t}]\!] \Rrightarrow T$ that relates a syntactic type `t` to a corresponding semantic type T. In particular, the (T-Rec) rule defines how a syntactic record type is related to a scope with a declaration for each field in the record type. We use the vector notation $\bar{x}$ to denote sequences and point-wise application. The mapping from syntactic to semantic types is used in the (ERS-Fun) rule (not shown) to convert the syntactic type annotation on the formal parameter. The (ERS-Rec) rule asserts that a record literal is typed by a scope that has a declaration for each field name in the list, inferring the type from the initialization expression. In the (T-Rec) and (ERS-Rec) rules we have omitted the assertion that field names of record types need to be unique. This can be expressed with a scope graph query that requires that a field name reference in the record scope resolves to a single declaration.

Composing Record Types Traditional type environments and scope graphs can be considered as a kind of *explicit substitution* (Abadi et al., 1991). The difference between the approaches is in their treatment of *extension* of substitutions. For example, consider the $\mathtt{e}_1$ **`extends`** $\mathtt{e}_2$ form, which creates a record by extending the record computed by $\mathtt{e}_2$ with the record fields computed by $\mathtt{e}_1$. In a variation on the definition by Pierce (2002), we allow a record extension to shadow fields from the extended record. Using type environments as record types, with the operator $\Gamma_1 \lhd \Gamma_2$ *eagerly* defined to compose two environments such that bindings in Γ_2 shadow those in Γ_1, the typing rule for **`extends`** can be defined as follows:

$$\text{(Γ-Extends)}\,\frac{\Gamma \vdash \mathtt{e}_1 : \{\mathtt{x}_i \mathtt{:} \mathtt{T}_i{}^{i\in 1..n}\} \qquad \Gamma \vdash \mathtt{e}_2 : \{\mathtt{x}_i \mathtt{:} \mathtt{T}_i{}^{j\in 1..m}\}}{\Gamma \vdash \mathtt{e}_1 \;\; \mathbf{\mathtt{extends}} \;\; \mathtt{e}_2 : \{\mathtt{x}_i \mathtt{:} \mathtt{T}_i{}^{j\in 1..m} \lhd \mathtt{x}_i \mathtt{:} \mathtt{T}_i{}^{i\in 1..n}\}}$$

So, the expression `{x=1, y=2}` **`extends`** `{x={z=4}}` has type `{x:num, y:num}`. A context resulting from a shadowing extension $\Gamma_1 \lhd \Gamma_2$ loses information about the structure of the original Γ_1, because it eagerly merges the two substitutions.

By contrast, a scope graph representation retains the structure of the composition. Consider rule (ERS-Extends) in Fig. 3.4, which defines the **`extends`** form by creating a new record type $\mathsf{REC}(s_r)$ with scope edges to the record types of the two branches. The `R` edge in (ERS-Extends) makes the bindings in the s_1 scope for the record ex-

```
let r1 = {a2 = 23, b3 = {...}} in
let q5 = {b6 = 19} extends r7 in
let f8 = fun(p9 : {b10 : num}){p11.b12} in
f13 q14 + q15.b16
```

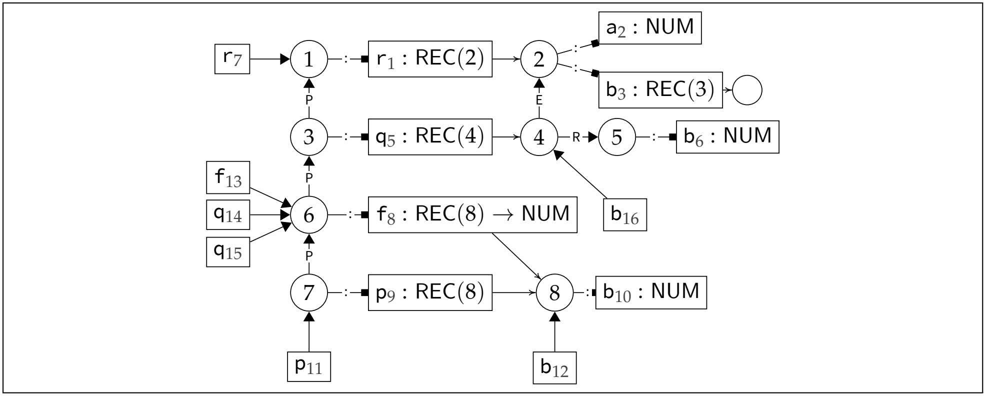


Figure 3.6: A program with records and functions

tension reachable from s_r. Similarly, the E edge makes the bindings in the s_2 scope reachable from s_r. Fig. 3.5 shows the resulting scope graph for the expression above. Thus, extensions are represented as edges that preserve the structure of the substitutions being merged.

The (ERS-With) rule shows a variation of this pattern. The form `with e1 do e2` (inspired by the deprecated JavaScript construct) makes the fields of the record computed by $\mathtt{e}_1$ available as local variables in $\mathtt{e}_2$. This is modeled by the (ERS-With) rule by creating scope edges from the scope s_w for the body $\mathtt{e}_2$ to the record scope via an R edge and to the lexical parent scope via a P edge.

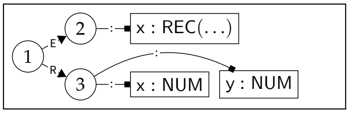


Figure 3.5: Record extension

Accessing Record Types Field access `e.x` is an example of *type-dependent name resolution* where a name is resolved relative to a type. The first premise of rule (ERS-Access) requires the expression e to have a record type $\mathsf{REC}(s_r)$. The second premise resolves the field x relative to the scope s_r of that type using the resolution query $\textsc{decl}(\mathtt{x}_i), (\mathsf{R}|\mathsf{E})^*, \leq_\top, <_l \vdash p : s_r \overset{\cdot}{\mapsto} \mathtt{x}_j : T$. The declaration well-formedness predicate $\textsc{decl}(\mathtt{x}_i)$ is defined in Fig. 3.3, and path well-formedness is given by a regular expression stating that resolution may follow any path via R and E edges. Record fields can also be accessed using plain variables due to the **with** form. Since variables may also be defined in lexical parents, the well-formedness for variable resolution first traverses a series of lexical parent edges before considering record (extension) edges: $\textsc{decl}(\mathtt{x}_i), (\mathtt{P}^*(\mathtt{R}|\mathtt{E})^*), \leq_\top, <_l \vdash p : s_r \overset{.}{\mapsto} \mathtt{x}_j : T$.

The visibility ordering $<_l$ in Fig. 3.4 states that record edges ($\mathtt{R}$) are preferred over both lexical parent edges ($\mathtt{P}$) and extension edges ($\mathtt{E}$). Consequently, declarations in record scopes shadow lexical bindings (as intended for **`with`** expressions), and extended record scope bindings (as intended for **`extends`** expressions). Consider the resolution of the field access $\mathtt{q}_{15}\texttt{.}\mathtt{b}_{16}$ in Fig. 3.6. The variable $\mathtt{q}_{15}$ is resolved relative to scope #6 to $\mathtt{q}_5$ with type $\mathsf{REC}(4)$. Hence, field $\mathtt{b}_{16}$ is resolved relative to scope #4 from which two declarations can be reached: $\mathtt{b}_3$ and $\mathtt{b}_6$. Since $\mathtt{R} <_l \mathtt{E}$, the latter is selected.

Comparing Record Types Finally, we consider the definition of subtyping for structural record types. When a record of type $\mathsf{REC}(s_2)$ is expected we may provide a record of type $\mathsf{REC}(s_1)$ provided that $\mathsf{REC}(s_1)$ has at least all the fields of $\mathsf{REC}(s_2)$. This is expressed using the resolution calculus by means of querying the visible fields of the scopes of the super type and sub type. The (<:-Rec) rule in Fig. 3.4 asserts that for each declaration $\mathtt{x}_i$ with type T visible in scope s_2 of the super type, $\mathtt{x}_i$ resolves to a declaration of type U in scope s_1 of the sub type, and that $U <: T$. The (<:-Rec) rule corresponds to traditional structural record subtyping (Pierce, 2002, Fig. 16-1). For example, consider the function application $\mathtt{f}_{13}\ \ \mathtt{q}_{14}$ in Fig. 3.6. $\mathtt{f}_{13}$ resolves to $\mathtt{f}_8$ with type $\mathsf{REC}(8) \rightarrow \mathsf{NUM}$ and $\mathtt{q}_{14}$ resolves to $\mathtt{q}_5$ with $\mathsf{REC}(4)$. The rule for function application (omitted) adapts (STLC-App) to require that the type of the actual parameter is a subtype of the type of the formal parameter. (This is the only rule using the subtype relation.) This is the case in our example since $\mathsf{REC}(4) <: \mathsf{REC}(8)$: the $\mathtt{b}_6$ field visible in scope #4 matches the $\mathtt{b}_{10}$ field of scope #8.

Summary A crucial difference between scope graphs and association lists is that association lists represent an *eager* name shadowing policy (applied before doing name resolution), while scope graphs support a *lazy* name shadowing policy (applied during name resolution). The scopes as types approach scales to type systems with binding patterns that go beyond lambdas and records, including type systems for languages with classes; association lists alone do not.

3.2.4 Classes and Nominal Subtyping: Featherweight Java

Next we consider a type system for *classes with subtyping*, specifically for *Featherweight Java (FJ)* (Igarashi et al., 2001). We show how nominal class identity and subtyping is characterized by scope identity and paths in the scope graph. The syntax and typing rules of FJ using scope graphs is summarized in Fig. 3.7 and 3.8. Assuming some familiarity with FJ, we summarize the main highlights.

Syntax

names	C, D, E, f, g, m, n, x	$\in$ *Name*	:=	some countable set
class definitions	L	$\in$ *ClassDecl*	::=	**class** C **extends** C {(C f;)* K M*}
constructors	K	$\in$ *KDecl*	::=	C((C f)*){**super**(f*);(**this**.f=f;)*}
methods	M	$\in$ *Methods*	::=	C m((C x)*){**return** e;}
expressions	e	$\in$ *Expr*	::=	x | e.f | e.m(e*) | **new** C(e*) | (C)e
semantic type	T, U, V	$\in$ *Type*	::=	INST(s)
method type	M	$\in$ *MethType*	::=	$T^* \rightarrow T$
class types	L	$\in$ *ClassType*	::=	CLASS(s)

Resolution Syntax Parameters

labels	l	$\in \mathcal{L}$	::=	P | S
relations	r	$\in \mathcal{R}$	::=	: | K
data terms	d	$\in \mathcal{D}$	::=	C : L | m : M | **this** : T | $x : T$ | $\bar{T}$

Figure 3.7: Syntax for Featherweight Java

Class Tables The original presentation of FJ relies on various data structures for name resolution, notably class tables, type contexts, and the AST of classes themselves. Names are mapped to class definitions via the class table. In turn, the class table is used in auxiliary relations that define how to retrieve association lists of names and types for class members, by traversing the AST of classes. Thus, classes are used as a data structure since they are not reducible to a simple association list representation. But the AST of FJ programs is not an ideal data structure for *reuse* to define name resolution for other languages with nominal subtyping. For such languages we would have to re-specify similar auxiliary relations to do name resolution using a different AST. We show how the definition of a class table data structure is subsumed by the use of scope graphs.

Syntactic and Semantic Types FJ has a single kind of syntactic type, namely class names ranged over by C. The corresponding semantic type of a class name C is an INST(s) type where s is the scope of the class declared as C. The (T-Class) rule in Fig. 3.8 translates a syntactic type to a semantic type by resolving the name in the lexical context by following a sequence of P edges. The "root" scope is similar to a *class table*: it binds all class declarations that a program defines and is a dominating lexical context for all classes in a program. Whereas INST(s) represents an *instance* of the class identified by scope s in the scope graph, the *class type* CLASS(s) represents the *definition* of the class s, and is the type of declarations in the "root" scope.

Class Typing The structure of a class is reflected in the scope graph. The (FJ-Class) rule declares the name of a class (C) as being typed by the scope that defines it (s_c) in

Syntactic to Semantic Typing $\boxed{\mathcal{G}, s \vdash [\![\mathtt{C}_i]\!] \Rrightarrow T}$

$$(\text{T-Class})\ \frac{\textsc{decl}(\mathtt{C}_i), \mathsf{P}^*, \leq_\top, <_l \vdash s \mathrel{\dot{\mapsto}} \mathtt{C}_j : \mathsf{CLASS}(s_c)}{s \vdash [\![\mathtt{C}_i]\!] \Rrightarrow \mathsf{INST}(s_c)}$$

Class Typing $\boxed{\mathcal{G}, s \vdash \mathtt{L}\ \textsc{ok}}$

$$(\text{FJ-Class})\ \frac{\begin{array}{cccc} s \overset{:}{\longrightarrow}\!\blacksquare\, \mathtt{C}_i : \mathsf{CLASS}(s_c) & s_c \xrightarrow{\mathsf{P}} s & \textsc{decl}(\mathtt{D}_j), \epsilon, \leq_\top, <_l \vdash p : s \mathrel{\dot{\mapsto}} \mathtt{D}_k : \mathsf{CLASS}(s_d) & \\ \nabla s_c & s_c \xrightarrow{\mathsf{S}} s_d & s_c \vdash \bar{\mathtt{C}_h}\ \bar{\mathtt{f}_j};\ \mathtt{K}\ \textsc{ok} & s_c \vdash \bar{\mathtt{M}_g}\ \textsc{ok} \end{array}}{s \vdash \mathtt{\mathbf{class}}\ \mathtt{C}_i\ \mathtt{\mathbf{extends}}\ \mathtt{D}_j\ \{\ \bar{\mathtt{C}_h}\ \bar{\mathtt{f}_j};\ \mathtt{K}\ \bar{\mathtt{M}_g}\ \}\ \textsc{ok}}$$

Field and Constructor Typing $\boxed{\mathcal{G}, s \vdash \bar{\mathtt{C}}\ \bar{\mathtt{f}};\ \mathtt{K}\ \textsc{ok}}$

$$(\text{FJ-FldK})\ \frac{\begin{array}{c} \textsc{decl}(\mathtt{C}_z), \mathsf{P}, \leq_\top, <_l \vdash s \mathrel{\dot{\mapsto}} \mathtt{C}_u : \mathsf{CLASS}(s) \quad s \vdash [\![\bar{\mathtt{D}_i}]\!] \Rrightarrow \bar{T} \quad \mathit{WFD}_\top, \mathsf{S}, \leq_\top, <_l \vdash p : s \overset{\mathsf{K}}{\longmapsto} \bar{T} \\ s \vdash [\![\bar{\mathtt{C}_x}]\!] \Rrightarrow \bar{U} \quad s \vdash [\![\bar{\mathtt{E}_k}]\!] \Rrightarrow \bar{V} \quad \bar{U} = \bar{V} \quad s \overset{:}{\longrightarrow}\!\blacksquare\, \bar{\mathtt{f}_m} : \bar{U} \quad s \overset{\mathsf{K}}{\longrightarrow}\!\blacksquare\, \bar{T}, \bar{U} \quad \bar{\mathtt{g}_j} \simeq \bar{\mathtt{g}_h} \end{array}}{s \vdash \bar{\mathtt{C}_x}\ \bar{\mathtt{f}_y};\ \mathtt{C}_z(\bar{\mathtt{D}_i}\ \bar{\mathtt{g}_j}, \bar{\mathtt{E}_k}\ \bar{\mathtt{f}_g})\{\mathtt{\mathbf{super}}(\bar{\mathtt{g}_h});\ \mathtt{\mathbf{this}}.\bar{\mathtt{f}_m}=\bar{\mathtt{f}_n}\}\ \textsc{ok}}$$

Method Typing $\boxed{\mathcal{G}, s \vdash \mathtt{M}\ \textsc{ok}}$

$$(\text{FJ-Method})\ \frac{\begin{array}{c} s \vdash [\![\bar{\mathtt{D}_k}]\!] \Rrightarrow \bar{T} \qquad s \vdash [\![\mathtt{C}_i]\!] \Rrightarrow T \qquad s \overset{:}{\longrightarrow}\!\blacksquare\, \mathtt{m}_j : \bar{T} \to T \qquad \nabla s_m \\ s_m \xrightarrow{\mathsf{P}} s \quad s_m \overset{:}{\longrightarrow}\!\blacksquare\, \bar{\mathtt{x}_g} : \bar{T} \quad s_m \overset{:}{\longrightarrow}\!\blacksquare\, \mathtt{\mathbf{this}} : \mathsf{INST}(s) \quad s_m \vdash \mathtt{e} : U \quad \vdash U <: T \\ \text{if } (\textsc{decl}(\mathtt{m}_j), \mathsf{S}^+, \leq_\top, <_l \vdash p : s \mapsto \mathtt{n}_h : \bar{V} \to V) \text{ then } \bar{T} = \bar{V} \text{ and } T = V \end{array}}{s \vdash \mathtt{C}_i\ \mathtt{m}_j(\bar{\mathtt{D}_k}\ \bar{\mathtt{x}_g})\ \{\ \mathtt{\mathbf{return}}\ \mathtt{e};\ \}\ \textsc{ok}}$$

Expression Typing $\boxed{\mathcal{G}, s \vdash \mathtt{e} : T}$

$$(\text{FJ-Var})\ \frac{\textsc{decl}(\mathtt{x}_i), \mathsf{P}^*\mathsf{S}^*, \leq_\top, <_l \vdash p : s \mathrel{\dot{\mapsto}} \mathtt{x}_j : T}{s \vdash \mathtt{x}_i : T} \qquad (\text{FJ-Field})\ \frac{\begin{array}{c} s \vdash \mathtt{e} : \mathsf{INST}(s_c) \\ \textsc{decl}(\mathtt{f}_i), \mathsf{S}^*, \leq_\top, <_l \vdash p : s_c \mathrel{\dot{\mapsto}} \mathtt{f}_j : T \end{array}}{s \vdash \mathtt{e.f}_i : T}$$

$$(\text{FJ-Invk})\ \frac{s \vdash \mathtt{e} : \mathsf{INST}(s_c) \quad \textsc{decl}(\mathtt{m}_i), \mathsf{S}^*, \leq_\top, <_l \vdash p : s_c \mathrel{\dot{\mapsto}} \bar{U} \to T \quad s \vdash \bar{\mathtt{e}} : \bar{V} \quad \vdash \bar{V} <: \bar{U}}{s \vdash \mathtt{e.m}_i(\bar{\mathtt{e}}) : T}$$

$$(\text{FJ-New})\ \frac{\begin{array}{c} \textsc{decl}(\mathtt{C}_i), \mathsf{P}^*, \leq_\top, <_l \vdash s \mapsto \mathtt{C}_j : \mathsf{CLASS}(s_c) \quad s \vdash \bar{\mathtt{e}} : \bar{T} \\ \mathit{WFD}_\top, \epsilon, \leq_\top, <_l \vdash p : s_c \overset{\mathsf{K}}{\longmapsto} \bar{U} \qquad \vdash \bar{T} <: \bar{U} \end{array}}{s \vdash \mathtt{\mathbf{new}}\ \mathtt{C}_i(\bar{\mathtt{e}}) : \mathsf{INST}(s_c)}$$

$$(\text{FJ-UCast})\ \frac{s \vdash \mathtt{e} : T \quad s \vdash [\![\mathtt{C}_i]\!] \Rrightarrow U \quad \vdash T <: U}{s \vdash (\mathtt{C}_i)\mathtt{e} : U} \qquad (\text{FJ-DCast})\ \frac{s \vdash \mathtt{e} : T \quad s \vdash [\![\mathtt{C}_i]\!] \Rrightarrow U \quad \vdash U <: T \quad U \neq T}{s \vdash (\mathtt{C}_i)\mathtt{e} : U}$$

$$(\text{FJ-Stupid})\ \frac{s \vdash \mathtt{e} : T \quad s \vdash [\![\mathtt{C}_i]\!] \Rrightarrow U \quad \vdash T \not<: U \quad \vdash U \not<: T \quad \textit{stupid warning}}{s \vdash (\mathtt{C}_i)\mathtt{e} : U}$$

Subtyping $\boxed{\mathcal{G} \vdash T <: T}$

$$(<:\text{-Class})\ \frac{\vdash p : s_1 \twoheadrightarrow s_2 \quad p \in \mathsf{S}^*}{\vdash \mathsf{INST}(s_1) <: \mathsf{INST}(s_2)}$$

Label Order and Data Well-Formedness $\boxed{\hat{l} <_l \hat{l}}$ $\boxed{d \in \textsc{decl}(\mathtt{x}_i)}$ $\boxed{d \in \mathit{WFD}_\top}$

$$\$ <_l \mathsf{P} \qquad \$ <_l \mathsf{S} \qquad \frac{\mathtt{x}_i \simeq \mathtt{x}_j}{(\mathtt{x}_i : T) \in \textsc{decl}(\mathtt{x}_j)} \qquad \frac{}{d \in \mathit{WFD}_\top}$$

Figure 3.8: Typing rules for Featherweight Java

the "root" scope of a program (s). The rule omits the assertion that field and record names are unique in a class. (The $\xrightarrow{:}\blacksquare$ relation is overloaded to associate names with *either* semantic types, class types, or method types. It is always clear from the context which kind of type a name is associated with.) The (FJ-FldK) rule asserts that fields and constructors are associated with class scopes, where the constructor parameter types are recorded using the relation $\xrightarrow{\mathsf{K}}\blacksquare$. To resolve the parameter types of a constructor we use a trivially true well-formedness predicate $\mathit{WFD}_{\top}$ in (FJ-FldK). The (FJ-Method) rule asserts that well-typed methods are associated with the class scope, and that overriding methods have the same type signature as the overridden methods in super classes.

Fig. 3.9 shows a program with four classes, and the scope graph of this program. Each class has a name that is typed as a $\mathsf{CLASS}(s)$ where s is the scope of the class. Class scopes have a declaration for each member. For example, A_1 is associated with the class scope that has a single declaration f_2 of type T (a semantic type of T). Class scopes are connected to the scope of their super class via an edge labeled S (for super) which makes the class members in super classes reachable via name resolution. S edges are the result of resolving the **extends** clauses of classes (FJ-Class). For example, the class scope for B is connected to the class scope of A because A_4 in the program resolves to A_1. (For brevity we have omitted the extends clause references from the scope graph diagram.) Thus scopes directly represent and expose the inheritance structure of classes.

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

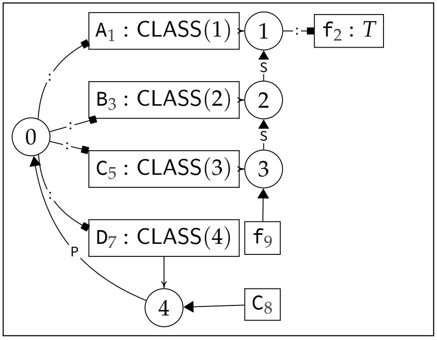


Figure 3.9: Classes with inheritance. The P edges from scopes #1, #2, #3 to scope #0 have been omitted.

Expression Typing The expression typing rules in Fig. 3.8 stay close to the original presentation of FJ by Igarashi et al. (2001); we discuss the generalizations we have made. The (FJ-Var) rule matches paths that either traverse a sequence of lexical parent edges, which makes formal parameters of methods *as well as* local fields reachable, or traverse a sequence of super edges which makes fields in super classes reachable. Thus, unlike the original presentation of FJ, field access need not happen via a qualified field access expression. The (FJ-New) rule for **new** expressions dereferences the constructor method of a class by resolving the $\xrightarrow{\mathsf{K}}\blacksquare$ relation in the class scope s_c; ϵ denotes the empty regular expression, which matches a 0-step path.

Subtyping Nominal subtyping allows the use of a sub-class in the place of any of its super classes: if A is a super type of B, then B can be used anywhere an A is expected. The scope graph affords a straightforward characterization of this subtype relationship: any class member declaration that is reachable from the class scope of A is also reachable from the inheriting class scope of B, because their scopes are connected via an S edge. In other words, using scopes as types lets us define nominal subtyping as path connectedness in a scope graph, as defined by (<:-Class) in Fig. 3.8.

3.2.5 Parametric Polymorphism

Parametric polymorphism characterizes types that are parameterized by other types and that can be instantiated by substitution. Thus to support parametric polymorphism when the structure of types is given by scopes, we need a notion of substitution over scopes in a scope graph. There are several ways to approach this task. A naive definition of a substitution function would eagerly traverse the structure of a scope graph to substitute named references that occur in the graph. Conceptually, this eager approach produces a new scope graph where some identifiers have been substituted. In other words, the approach *duplicates* parts of the scope graph. Our goal is to support the implementation of practical type checkers, so we prefer a substitution strategy that does *not* require inefficient duplication of scopes and scope graphs.

We present an approach based on scopes with *explicit substitutions* that are lazily applied during name resolution, as opposed to eager application before name resolution. We illustrate the approach with a specification of the type system of System F in Fig. 3.10. System F extends the simply-typed lambda calculus with explicitly parameterized types, type quantification expressions, and type application expressions. With the exception of parameterized types (`X => t`), the types in System F are rather simple and absent of name binding. As such it is not a language where scopes are an obviously well-suited choice of representation for types. Yet the same pattern of type parameterization occurs in languages with more interesting types, such Featherweight Generic Java (FGJ), the extension of Featherweight Java with *generics* (Igarashi et al., 2001). We use System F as an example language which illustrates the approach to parametric polymorphism using scopes as types, and discuss how this approach scales to FGJ.

Syntactic and Semantic Types There are two new kinds of syntactic and semantic types in Fig. 3.10, as compared with Fig. 3.3. Syntactically, `X => t` denotes a *forall type* that quantifies a type `t` over another type, ranged over by the named parameter `X`. The corresponding semantic $\mathsf{ALL}(\mathtt{X}, s)$ type quantifies *a scope* over a type. The (T-All) rule in Fig. 3.10 asserts that the scope s_a of a semantic forall type is: (1) connected to the lexical context scope s; (2) associated with the declared type variable using

Syntax

type identifiers $\mathsf{X} \in \mathit{TypeId} :=$ some countable set

expressions $\mathtt{e} \in \mathit{Expr} ::= \ldots \mid \mathtt{\textbf{Fun}(X)\{\ e\ \}} \mid \mathtt{e\ [t]}$

syntactic types $\mathtt{t} \in \mathit{Type} ::= \ldots \mid \mathtt{X\ {=}{>}\ t} \mid \mathtt{X}$

semantic types $T, U, V \in \mathit{Type} ::= \ldots \mid \mathsf{ALL}(\mathsf{X}, s) \mid \mathsf{X} \mid \pi_{\mathsf{B}}(s)$

Resolution Syntax Parameters

labels $l \in \mathcal{L} ::= \ldots \mid \mathsf{I}$ and otherwise like STLC

Syntactic to Semantic Typing $\boxed{\mathcal{G}, s \vdash [\![\mathtt{t}]\!] \Rrightarrow T}$

$$\text{(T-All)}\,\dfrac{\begin{array}{c}\nabla s_a \quad s_a \xrightarrow{\mathsf{P}} s \quad s_a \xrightarrow{\mathsf{V}}\!\blacksquare\ \mathsf{X}_i \\ s_a \vdash [\![\mathtt{t}]\!] \Rrightarrow T \quad s_a \xrightarrow{\mathsf{B}}\!\blacksquare\ T\end{array}}{s \vdash [\![\mathsf{X}_i\ \mathtt{=>}\ \mathtt{t}]\!] \Rrightarrow \mathsf{ALL}(\mathsf{X}_i, s_a)} \qquad \text{(T-Var)}\,\dfrac{\textsc{decl}(\mathsf{X}_i), \mathsf{P}^*, \leq_\top, <_l \vdash p : s \xmapsto{\mathsf{V}} \mathsf{X}_j}{s \vdash [\![\mathsf{X}_i]\!] \Rrightarrow \mathsf{X}_j}$$

Expression Typing (Selected Rules) $\boxed{\mathcal{G}, s \vdash \mathtt{e} : T}$

$$\text{(F-All)}\,\dfrac{\begin{array}{c}\nabla s_a \quad s_a \xrightarrow{\mathsf{P}} s \quad s_a \xrightarrow{\mathsf{V}}\!\blacksquare\ \mathsf{X}_i \\ s_a \vdash \mathtt{e} : T \quad s_a \xrightarrow{\mathsf{B}}\!\blacksquare\ T\end{array}}{s \vdash \mathtt{\textbf{Fun}(}\mathsf{X}_i\mathtt{)\ \{\ e\ \}} : \mathsf{ALL}(\mathsf{X}_i, s_a)} \qquad \text{(F-TApp)}\,\dfrac{\begin{array}{c} s \vdash \mathtt{e} : \mathsf{ALL}(\mathsf{X}_i, s_a) \quad s \vdash [\![\mathtt{t}]\!] \Rrightarrow T \\ \nabla s_k \quad s_k \xrightarrow{\mathsf{I}} s_a \quad s_k \xrightarrow{\sigma}\!\blacksquare\ \mathsf{X}_i := T\end{array}}{s \vdash \mathtt{e\ [t]} : \pi_{\mathsf{B}}(s_k)}$$

$$\text{(F-Strict)}\,\dfrac{s \vdash \mathtt{e} : T \quad \vdash T \Rightarrow U}{s \vdash \mathtt{e} : U}$$

Type Normalization $\boxed{\mathcal{G} \vdash T \Rightarrow T}\ \boxed{\mathcal{G}, p \vdash T \dot{\Rightarrow} T}$

$$\text{(Strict-Pi)}\,\dfrac{\begin{array}{c} \mathit{WFD}_\top, \mathsf{I}^*, \leq_\top, <_l \vdash p : s \xmapsto{\mathsf{B}} T \\ p \vdash T \dot{\Rightarrow} U \end{array}}{\vdash \pi_{\mathsf{B}}(s) \Rightarrow U} \qquad \text{(Strict-NotPi)}\,\dfrac{T \neq \pi_{\mathsf{B}}(s)}{\vdash T \Rightarrow T} \qquad \text{(N-Pi)}\,\dfrac{\vdash \pi_{\mathsf{B}}(s) \Rightarrow T \quad p \vdash T \dot{\Rightarrow} U}{p \vdash \pi_{\mathsf{B}}(s) \dot{\Rightarrow} U}$$

$$\text{(N-Done)}\,\dfrac{}{s \vdash T \dot{\Rightarrow} T} \qquad \text{(N-Num)}\,\dfrac{}{p \vdash \mathsf{NUM} \dot{\Rightarrow} \mathsf{NUM}} \qquad \text{(N-Fun)}\,\dfrac{p \vdash T_1 \dot{\Rightarrow} T_2 \quad p \vdash U_1 \dot{\Rightarrow} U_2}{p \vdash T_1 \to U_1 \dot{\Rightarrow} T_2 \to U_2}$$

$$\text{(N-All)}\,\dfrac{\begin{array}{c} \mathit{WFD}_\top, \epsilon, \leq_\top, <_l \vdash p' : s_k \xmapsto{\sigma} \mathsf{X}_j := T \quad s'_k \xrightarrow{\mathsf{I}} s_a \\ \nabla s'_k \quad p \vdash \mathsf{ALL}(\mathsf{X}_i, s'_k) \dot{\Rightarrow} T \quad s'_k \xrightarrow{\sigma}\!\blacksquare\ \mathsf{X}_j := T \end{array}}{p \cdot \mathsf{I} \cdot s_k \vdash \mathsf{ALL}(\mathsf{X}_i, s_a) \dot{\Rightarrow} T} \qquad \text{(N-Var)}\,\dfrac{\begin{array}{c} \mathit{WFD}_\top, \epsilon, \leq_\top, <_l \vdash p' : s_k \xmapsto{\sigma} \mathsf{X}_j := T \\ \text{if } \mathsf{X}_i = \mathsf{X}_j \text{ then } U = T \text{ else } U = \mathsf{X}_i \\ p \vdash U \dot{\Rightarrow} V \end{array}}{p \cdot \mathsf{I} \cdot s_k \vdash \mathsf{X}_i \dot{\Rightarrow} V}$$

Semantic Type Equality $\boxed{\mathcal{G} \vdash T \cong T}$

$$\text{(Eq-Num)}\,\dfrac{}{\vdash \mathsf{NUM} \cong \mathsf{NUM}} \qquad \text{(Eq-Fun)}\,\dfrac{\vdash T_1 \cong T_2 \quad \vdash U_1 \cong U_2}{\vdash T_1 \to U_1 \cong T_2 \to U_2} \qquad \text{(Eq-Var)}\,\dfrac{\mathsf{X}_i = \mathsf{X}_j}{\vdash \mathsf{X}_i \cong \mathsf{X}_j}$$

$$\text{(Eq-All)}\,\dfrac{\begin{array}{c} \nabla s'_1 \quad s'_1 \xrightarrow{\mathsf{I}} s_1 \quad s'_1 \xrightarrow{\sigma}\!\blacksquare\ \mathsf{X}_i := \mathsf{X} \\ \nabla s'_2 \quad s'_2 \xrightarrow{\mathsf{I}} s_2 \quad s'_2 \xrightarrow{\sigma}\!\blacksquare\ \mathsf{X}_j := \mathsf{X} \\ \vdash \pi_{\mathsf{B}}(s'_1) \cong \pi_{\mathsf{B}}(s'_2) \qquad \nabla \mathsf{X} \end{array}}{\vdash \mathsf{ALL}(\mathsf{X}_i, s_1) \cong \mathsf{ALL}(\mathsf{X}_j, s_2)} \qquad \text{(Eq-Pi1)}\,\dfrac{\begin{array}{c} \vdash \pi_{\mathsf{B}}(s) \Rightarrow U \\ \vdash U \cong T \end{array}}{\vdash \pi_{\mathsf{B}}(s) \cong T} \qquad \text{(Eq-Pi2)}\,\dfrac{\begin{array}{c} \vdash \pi_{\mathsf{B}}(s) \Rightarrow U \\ \vdash T \cong U \end{array}}{\vdash T \cong \pi_{\mathsf{B}}(s)}$$

Label Order $\boxed{\hat{l} <_l \hat{l}}$

$$\mathsf{\$} <_l \mathsf{P} \qquad \mathsf{\$} <_l \mathsf{I}$$

Figure 3.10: Syntax and typing rules for System F (expressions and syntactic types extend Fig. 3.3)

the $\xrightarrow{\mathsf{V}}\!\!\blacksquare$ relation; and (3) associated with the semantic type in the body of the forall type via the $\xrightarrow{\mathsf{B}}\!\!\blacksquare$ relation. Semantic forall types are reminiscent of how parameterized types are represented in the Dependent Object Types (DOT) calculus (Amin et al., 2016; Amin and Rompf, 2017), where a parameterized type can be represented as a two-field record with an abstract type field ($\xrightarrow{\mathsf{V}}\!\!\blacksquare$), and another field whose type may contain named references to the abstract type field. (Indeed, the scopes-as-types approach was inspired by the treatment of type parameters in DOT.)

Rule (T-Var) defines the semantic type of a type variable reference $\mathtt{X}_i$ to be the type variable declaration $\mathtt{X}_j$ that the reference resolves to and that uniquely identifies a declared type parameter.

Expression Typing Fig. 3.10 summarizes the typing rules for the syntactic forms that introduce forall types (**`Fun`**`(`$\mathtt{X}_i$`){ e }`) and eliminate forall types (`e [t]`). The introduction rule (F-All) is similar to the (T-All) rule. The (F-TApp) rule asserts that there is an *instantiation scope* s_k with an *explicit substitution* of the parameter $\mathtt{X}_i$ by the argument type T. This instantiation scope is associated with the scope of the forall type via an instantiation edge $\xrightarrow{\mathsf{I}}\!\!\blacksquare$. Instead of eagerly propagating the explicit substitution, the (F-TApp) rule returns a type $\pi_{\mathsf{B}}(s)$ representing a delayed projection of the body of a semantic forall type. When needed, we apply *strictness* (discussed below) to normalize projections. Not shown in Fig. 3.10 are the rules for the STLC fragment of System F. The only difference from Section 3.2.2 is that function application uses *semantic type equality*, which we also discuss below, to require that the type of the actual parameter matches the type of the formal parameter.

Type Normalization Strictness $\vdash T \Rrightarrow U$ forces the application of delayed projections that occur in the head position of T to obtain a normalized type U. Projections are applied by using the resolution calculus in (Strict-Pi) to resolve the nearest $\xrightarrow{\mathsf{B}}\!\!\blacksquare$ relation through a sequence of instantiation scopes (which correspond to delayed and explicit substitutions), and then normalizing the resolved type with respect to each instantiation scope.

The (N-Done) rule matches on a path consisting of a single scope, that is, a 0-step path. The two most interesting rules for normalization are the (N-All) rules and the (N-Var) rules. The (N-All) rule normalizes a forall type by matching on a path in reverse order (i.e., the order in which sequenced instantiation scopes have been created), to augment the scope of a forall-type with each explicit substitution found along the projection path. The (N-Var) rule also matches on paths in reverse order and resolves the substitution in the instantiation scope s_k. The substitution is only applied if the resolved substitution is for a type variable parameter $\mathtt{X}_j$ that is *syntactically equal* to the variable $\mathtt{X}_i$ being normalized; that is, the position subscripts on the identifiers *must match*. Because we use the declaration identifiers as the semantic type of type variable references, we avoid problems with shadowing and name capture. Consider, for example, how type normalization applies to the term

$$\texttt{(}\mathbf{Fun}\texttt{(}\mathtt{A}_1\texttt{) \{ }\mathbf{Fun}\texttt{(}\mathtt{A}_2\texttt{)\{ }\mathbf{fun}\texttt{(}\mathtt{x}_3\texttt{:}\mathtt{A}_4\texttt{)\{ }\mathtt{x}_5\texttt{ \} \} \}) [}\mathbf{num}\texttt{].}$$

The substitution $\mathtt{A}_1 := \mathbf{num}$ will be recorded in the semantic forall type that is returned, but will never substitute the semantic type of the reference in the innermost **Fun** (i.e., $\mathtt{A}_2$) because it has a different position subscript.

Semantic Type Equality Fig. 3.10 also defines a notion of semantic type equality between semantic types. The most interesting rule is the (Eq-All) rule for forall types. The premises of this rule assert that we create instantiation scopes which substitute the parameter names by *the same* identifier X where X is chosen to be fresh. We then compare the result of projecting the body of the semantic forall types in the context of these instantiation scopes. This parameter instantiation makes alpha-equivalent forall types match. The (Eq-Var) rule equates type variables by using syntactic equality. Projections are compared by applying strictness as defined by the rules (Eq-Pi1) and (Eq-Pi2).

From System F to Generic Classes in FGJ The typing rules in Fig. 3.10 define an approach to substitution in scopes that does not require inefficient duplication of scopes and scope graphs. Instead of eagerly propagating substitutions, which result in duplicating scope graphs, we record delayed and explicit substitutions in the scope graph, thereby *sharing* scopes between different type parameter instantiations. This approach scales to languages where types have interesting binding structure, such as Featherweight Java with generic classes, FGJ. For brevity, we omit the full specification of the type system for FGJ and instead discuss an illustrative example program and its corresponding scope graph diagram. The artifact accompanying this paper contains implementations of type checkers for both System F and FGJ in Statix.

$$\begin{array}{l}
\mathbf{class}\ \mathtt{A}_1\texttt{<}\mathtt{X}_2\texttt{> \{ }\mathtt{X}_3\ \mathtt{f}_4\texttt{; \}}\\
\texttt{...}\\
\mathtt{m}_5\texttt{ = }\mathbf{new}\ \mathtt{A}_6\texttt{<T>();}\\
\mathtt{m}_7\texttt{.}\mathtt{f}_8\texttt{;}\\
\mathtt{n}_9\texttt{ = }\mathbf{new}\ \mathtt{A}_{10}\texttt{<S>();}\\
\mathtt{n}_{11}\texttt{.}\mathtt{f}_{12}\texttt{;}
\end{array}$$

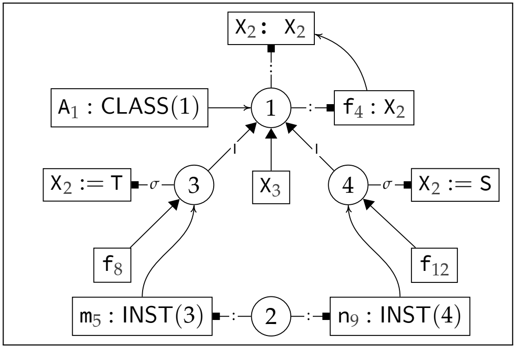


Figure 3.11: Generic class with two instantiations

Fig. 3.11 shows a program with a class definition A with a type parameter X and a single field f, typed with the type parameter X. The program also contains two instantiations of A with different type arguments. The field accesses $\mathtt{m}_7.\mathtt{f}_8$ and $\mathtt{n}_9.\mathtt{f}_{10}$ both resolve to the field in A. However, their type should be considered relative to the specific instantiation of the type parameter. That is, $\mathtt{m}_7.\mathtt{f}_8$ has type T and $\mathtt{n}_9.\mathtt{f}_{10}$ has type S (for some types T and S).

The scope graph in Fig. 3.11 illustrates how generic class instantiation is modeled using scope graphs: each generic class instantiation is modeled as an instantiation scope (scopes #3 and #4 in the figure). The instantiation $\mathtt{m}_5$ = **new** $\mathtt{A}_6$<T>() gives rise to scope #3 with the substitution $\mathtt{X}_2 := \mathtt{T}$. As in System F, delayed substitutions are applied to field types once a *field is accessed*, as opposed to eagerly when the class is initialized. By delaying the substitution as an instantiation scope we save having to duplicate the entire class scope when we instantiate the generic class A with a different generic type argument S. The class members of the class scope for A (scope #1) remain reachable via the I-labeled instantiation edge between scope #3 and scope #1.

3.2.6 Discussion

As argued above, scope graphs provide a data structure for name binding and resolution that does not prematurely optimize for particular binding patterns. We have shown that scope graphs can deal with type systems with parametric polymorphism in a way that also does not prematurely optimize for particular binding patterns. By recording substitutions explicitly in the scope graph we retain a history of substitutions to be applied to a type, and only *during resolution* of a particular relation do we actually apply the substitutions. This avoids duplication of scope graphs, and makes the approach promising for languages that do normalization during type checking for types with rich binding structure. It also shows that scope graphs and the revised resolution calculus presented in Section 3.2.1 provide a theory for name binding and name resolution in type systems that scales to languages beyond the relatively simple type systems that scope graphs were demonstrated to work previously (Bach Poulsen et al., 2016, 2018; Néron et al., 2015; van Antwerpen et al., 2016).

The notions of normalization and semantic type equality in Fig. 3.10 are inductively defined over the syntax of types, which is language specific. Our goal with scope graphs is to develop tools that are *reusable* between different languages. From this perspective it is not ideal that type normalization and semantic type equality is defined in a language-specific way. The notions of type normalization and semantic type equality that we have defined for System F and FGJ follow a similar pattern which indicates the existence of a schema for automatically generating notions of strictness and type equality. An alternative would be to augment the resolution calculus to support applying substitutions along a path.

Typing rules that use scope graphs are close to traditional type system rules such as those found in textbooks like Pierce's (2002). Some rules that use scope graphs are less concise than traditional rules due to the explicit passing of parameters to the resolution calculus, but we argue that this source of verbosity is outweighed by the benefits afforded by scope graphs: uniform treatment of name binding that does not prematurely optimize for particular binding patterns. The distinction between syntactic and semantic types found in type systems using scopes as types is rarely made in traditional type system specifications, although it is not uncommon in type system *implementations* where, for example, de Bruijn indices are commonly used to represent bindings in types. Formal definitions of type equality and substitution are commonly omitted from traditional type system specifications by alluding to the existence of a "standard" substitution function and alpha renaming scheme. Nevertheless, type system implementations must implement these notions. Thus the additional (as compared with traditional type system specification) rules for syntactic to semantic typing, type equality, and type substitution all help bridge the gap between type system specification and type system implementation, which is the goal that this work is pursuing.

3.3 Statix: Specification with Scopes and Constraints

Type systems written in the style of the previous section do not immediately give us executable implementations. In this section we introduce Statix, a specification language to develop type checkers with scope graphs, which has precise declarative and operational interpretations. Rules in Statix are close to the inference rules of the previous section, but the language makes several finer points of those rules, that we glossed over before, precise. We explain the language using an example specification and define a formal declarative semantics.

Statix by Example The formal syntax of Statix is defined in Fig. 3.13. A Statix program consists of a collection of user-defined constraint rules, together with a top-level constraint. Rules must be syntax-directed, with non-overlapping guards, and are expressed in terms of equality, scope graph based name resolution, and user-defined constraints. We introduce the language using the constraint rules in Fig. 3.12, which define the simply-typed lambda calculus of Fig. 3.3.

The typing relation $s \vdash \mathsf{e} : \mathsf{T}$ is expressed as the user-defined constraint

$$\mathsf{typeOfExp}(s, \mathsf{e}, \mathsf{T})$$

Analogous to Fig. 3.3, Fig. 3.12 defines a rule for each expression form. A rule of the form $c(\bar{t}) \leftarrow C$ states that a constraint matching the *head* $c(\bar{t})$ holds, if the *body* constraint C holds. Constraints are combined using conjunction $C \wedge C$. The body of a rule may invoke user-defined constraints by applying the predicate name to a list of terms $c(\bar{t})$. For example, the rule

$$\mathsf{typeOfExp}(s, \mathsf{e}_1 \texttt{ + } \mathsf{e}_2, \mathsf{T}) \leftarrow \mathsf{T} = \mathbf{num} \wedge \mathsf{typeOfExp}(s, \mathsf{e}_1, \mathbf{num}) \wedge \mathsf{typeOfExp}(s, \mathsf{e}_2, \mathbf{num})$$

$$
\begin{aligned}
\mathsf{typeOfExp}(s, z, \mathtt{T}) &\leftarrow \mathtt{T} = \mathbf{num} \\
\mathsf{typeOfExp}(s, \mathtt{e_1\ +\ e_2}, \mathtt{T}) &\leftarrow \mathtt{T} = \mathbf{num} \wedge \mathsf{typeOfExp}(s, \mathtt{e_1}, \mathbf{num}) \wedge \\
&\qquad \mathsf{typeOfExp}(s, \mathtt{e_2}, \mathbf{num}) \\
\mathsf{typeOfExp}(s, \mathbf{fun}\mathtt{(x}_i\mathtt{:T_1)\{\ e\ \}}, \mathtt{T}) &\leftarrow \exists \mathtt{T_2}.\, \exists s_{\mathrm{f}}.\, \mathtt{T} = \mathtt{FUN(T_1,\ T_2)} \wedge \nabla s_{\mathrm{f}} \wedge s_{\mathrm{f}} \xrightarrow{\mathsf{P}} s \wedge \\
&\qquad s_{\mathrm{f}} \xrightarrow{:} (\mathtt{x}_i, \mathtt{T_1}) \wedge \mathsf{typeOfExp}(s_{\mathrm{f}}, \mathtt{e}, \mathtt{T_2}) \\
\mathsf{typeOfExp}(s, \mathtt{x}_i, \mathtt{T}) &\leftarrow \mathtt{x}_i \text{ in } s \overset{:}{\mapsto} (\mathtt{x}_j, [\mathtt{T}|[]]) \\
\mathsf{typeOfExp}(s, \mathtt{e_1\ e_2}, \mathtt{T_2}) &\leftarrow \exists \mathtt{T_1}.\, \mathsf{typeOfExp}(s, \mathtt{e_1}, \mathtt{FUN(T_1,\ T_2)}) \wedge \\
&\qquad \mathsf{typeOfExp}(s, \mathtt{e_2}, \mathtt{T_1}) \\
\mathsf{typeOfExp}(s, \mathbf{let}\ \mathtt{x=e_1}\ \mathbf{in}\ \mathtt{e_2}, \mathtt{T_2}) &\leftarrow \exists \mathtt{T_1}.\, \exists s_{\mathrm{b}}.\, \mathsf{typeOfExp}(s, \mathtt{e_1}, \mathtt{T_1}) \wedge \nabla s_{\mathrm{b}} \wedge s_{\mathrm{b}} \xrightarrow{\mathsf{P}} s \wedge \\
&\qquad s_{\mathrm{b}} \xrightarrow{:} (\mathtt{x}_i, \mathtt{T_1}) \wedge \mathsf{typeOfExp}(s_{\mathrm{b}}, \mathtt{e_2}, \mathtt{T_2})
\end{aligned}
$$

Figure 3.12: Statix specification for a simply-typed lambda calculus using scope graphs

uses typeOfExp to constrain the types of the sub-expressions. All variables matched in the head are bound in the body of a rule. Local variables are introduced using $\exists v.C$. For example, the rule for **fun** introduces local variables for the return type $\mathtt{T_2}$ and the function scope s_{f}.

For ease of reading, we define each predicate using a set of rules and inline matches in the rule head. This desugars to a single rule using *guarded choice* $G\ ?\ C\ :\ C$ in the formal syntax:

$$\mathsf{typeOfExp}(s, \mathtt{e}, \mathtt{T}) \leftarrow (\mathtt{e} = z\ ?\ \mathtt{T} = \mathbf{num}\ :\ (\mathtt{e} = \mathtt{e_1\ +\ e_2}\ ?\ \ldots\ :\ (\ldots\ ?\ \ldots\ :\ \bot)))$$

Guarded choice is *committed* choice: for $G\ ?\ C_1\ :\ C_2$, either G and C_1 hold, or G does not hold and C_2 holds. Thus, if G holds, C_2 is never considered. Guards are restricted to existential quantification and term equality, to ease reasoning about coverage and non-overlapping rules.

Syntactic equality is expressed with the *equality* constraint $t_1 = t_2$. For example, it is used to constrain the type to **num** in the rules for z and $+$. Note that these types are written inline in the judgments of Fig. 3.3. Logically, they are equivalent, but, operationally it matters whether terms appear in the body, or in the head, where they are used for rule selection (see Section 3.4.1).

Three constraints assert facts about the scope graph. The constraint ∇t (pronounced: t is *fresh*) is satisfied if the scope value t is different from scope values t' that appear as $\nabla t'$ elsewhere. As such, one can think of this as claiming exclusive ownership of the scope. An *edge* constraint $t_1 \xrightarrow{l} t_2$ asserts the existence of an l edge from t_1 to t_2. Similarly, a data constraint $t_1 \xrightarrow{r} t_2$ asserts the existence of a t_2 value in the relation r in scope t_1. Using these constraints, the rules in Fig. 3.12 for function and let expressions specify their local scope, its parent edge, and the binding declaration.

Signature

$$\begin{array}{rrcll} \text{function symbols} & f,g & \in & \mathcal{F} & \text{with arity}(f) \in \mathbb{N} \\ \text{predicates symbols} & c,d & \in & \mathcal{C} & \text{with arity}(c) \in \mathbb{N} \end{array}$$

Definitions

$$\begin{array}{rrclll} \text{term variables} & x & \in & \mathcal{V} & := & \text{some countable set} \\ \text{terms} & t,u & \in & \mathcal{T} & ::= & x \mid f(\bar{t}) \mid s \mid p \mid l \mid [] \mid [t|t] \mid (t,t) \\ \text{guards} & G & \in & \mathit{Guards} & ::= & \top \mid G \wedge G \mid t = t \mid \exists x.G \\ \text{constraints} & C & \in & \mathcal{C} & ::= & \top \mid \bot \mid t = t \mid C \wedge C \mid c(\bar{t}) \mid \exists x.C \mid G \mathbin{?} C \ :\ C \\ & & & & \mid & \nabla t \mid t \xrightarrow{l} t \mid t \overset{r}{-\!\!\blacksquare}\, t \\ & & & & \mid & \text{query } \ (c(\bar{t}), c(\bar{t}), c(\bar{t}), c(\bar{t})) \ \text{ in } t \overset{r}{\mapsto} t \mid t \backsim^{\mathsf{P}} \mathit{re} \\ \text{predicates} & \mathit{pred} & \in & \mathit{Preds} & ::= & c(\bar{y}) \leftarrow C \\ \text{program} & P & \in & \mathit{Progs} & ::= & \text{let } \mathit{pred}^* \text{ in } C \end{array}$$

Figure 3.13: Syntax of Statix

Definitions substitution $\varphi, \theta : \mathcal{V} \to t$ scope support $S \subseteq \mathcal{S}$

Constraint satisfaction $\boxed{\mathcal{G}, \varphi \models_S C}$

$$\text{(DS-True)}\,\frac{}{\mathcal{G}, \varphi \models_S \top} \qquad \text{(DS-Eq)}\,\frac{t_1\varphi = t_2\varphi}{\mathcal{G}, \varphi \models_S t_1 = t_2} \qquad \text{(DS-Conj)}\,\frac{\mathcal{G}, \varphi \models_{S_1} C_1 \quad \mathcal{G}, \varphi \models_{S_2} C_2}{\mathcal{G}, \varphi \models_{S_1 \sqcup S_2} C_1 \wedge C_2}$$

$$\text{(DS-Disj-L)}\,\frac{\mathcal{G}, \varphi \models_\emptyset G \quad \mathcal{G}, \varphi \models_S C_1}{\mathcal{G}, \varphi \models_S G \mathbin{?} C_1 \ :\ C_2} \qquad \text{(DS-Disj-R)}\,\frac{\mathcal{G}, \varphi \not\models_\emptyset G \quad \mathcal{G}, \varphi \models_S C_2}{\mathcal{G}, \varphi \models_S G \mathbin{?} C_1 \ :\ C_2}$$

$$\text{(DS-Pred)}\,\frac{(c(\bar{x}) \leftarrow C) \in P \quad \mathcal{G}, \varphi \models_S C\,[\bar{t}/\bar{x}]}{\mathcal{G}, \varphi \models_S c(\bar{t})} \qquad \text{(DS-RegExp)}\,\frac{t\varphi = \bar{l} \quad \bar{l} \text{ in language of } \mathit{re}}{\mathcal{G}, \varphi \models_S t \backsim^{\mathsf{P}} \mathit{re}}$$

$$\text{(DS-Exists)}\,\frac{\mathcal{G}, \varphi[t/x] \models_S C \quad x \text{ is fresh for } \varphi}{\mathcal{G}, \varphi \models_S \exists x.C} \qquad \text{(DS-Fresh)}\,\frac{t\varphi = s \quad s \in S}{\mathcal{G}, \varphi \models_S \nabla t}$$

$$\text{(DS-Edge)}\,\frac{\begin{array}{c} t_1\varphi = s_1 \qquad t_2\varphi = s_2 \\ (s_1 \xrightarrow{l} s_2) \in \mathsf{edges}(\mathcal{G}) \end{array}}{\mathcal{G}, \varphi \models_S t_1 \xrightarrow{l} t_2} \qquad \text{(DS-Rel)}\,\frac{t_1\varphi = s \quad (s \overset{r}{-\!\!\blacksquare}\, t_2\varphi) \in \mathsf{data}(\mathcal{G})}{\mathcal{G}, \varphi \models_S t_1 \overset{r}{-\!\!\blacksquare}\, t_2}$$

(DS-Resolve)

$$\frac{\begin{array}{cc} \mathit{WFD} := \left\{ d \mid \mathcal{G}, \varphi \models_\emptyset c_{\mathit{wfd}}(u_1, ..., u_n, d) \right\} & \mathit{WFL} := \left\{ \bar{l} \mid \mathcal{G}, \varphi \models_\emptyset c_{\mathit{wfl}}(u'_1, ..., u'_n, \bar{l}) \right\} \\ d \leq_d d' := \mathcal{G}, \varphi \models_\emptyset c_{\leq d}(v_1, ..., v_m, d, d') & l <_l l' := \mathcal{G}, \varphi \models_\emptyset c_{<l}(v'_1, ..., v'_m, l, l') \\ \multicolumn{2}{c}{t_1\varphi = s \quad (p,t) \in t_2\varphi \iff \left(\mathit{WFD}, \mathit{WFL}, \leq_d, <_l, \mathcal{G} \vdash p : s \overset{r}{\mapsto} t \right) \quad \textit{no duplicates in } t_2\varphi} \end{array}}{\mathcal{G}, \varphi \models_S \text{query } \left(c_{\mathit{wfd}}(u_1, ..., u_n), c_{\mathit{wfl}}(u'_1, ..., u'_n), c_{\leq d}(v_1, ..., v_m), c_{<l}(v'_1, ..., v'_m) \right) \text{ in } t_1 \overset{r}{\mapsto} t_2}$$

Figure 3.14: Declarative semantics of Statix

Finally, *resolution* constraints specify queries on the scope graph. A resolution constraint $\mathsf{query}\ \left(c_{wfd}(\bar{u}), c_{wfl}(\bar{u}), c_{\leq d}(\bar{u}), c_{<l}(\bar{u})\right)\ \mathsf{in}\ t_1 \xrightarrow{r} t_2$ states that resolving the relation r in scope t_1, results in t_2. The well-formedness and order predicates correspond to the parameters of the resolution calculus defined in Fig. 3.1. The well-formedness and order predicates can be partially applied, to make resolution context aware.

For example, the variable rule uses the short-hand notation $x_i\ \mathsf{in}\ s \stackrel{\cdot}{\mapsto} (x_j, \mathsf{T})$ for resolving data from a reference occurrence, which corresponds to

$$\mathsf{query}\ \left(\mathsf{wfd}(x_i), \mathsf{wfl}, \mathsf{ord}_d, \mathsf{ord}_l\right)\ \mathsf{in}\ s \stackrel{\cdot}{\mapsto} (x_j, \mathsf{T}).$$

The data well-formedness is partially applied to the reference x_i, and the result of resolution must be a single declaration-type pair. The label well-formedness wfl and data well-formedness wfd are defined with the rules $\mathsf{wfl}(x_i, ls) \leftarrow ls \backsim^{\mathsf{P}} \mathsf{P}^*$ and $\mathsf{wfd}(x_i, y_j) \leftarrow x = y$. The label order is defined in terms of the *match* constraint $t \backsim^{\mathsf{P}} re$, which states that the list of labels t must match the regular expressions re. The label order ord_l and data order ord_d define a lexical ordering with the rules $\mathsf{ord}_l(\$, \mathsf{P}) \leftarrow \top$ and $\mathsf{ord}_d(x_i, y_j) \leftarrow \top$.

Declarative Semantics The declarative semantics in Fig. 3.14 gives a precise definition of the meaning of the constraints in terms of a satisfaction relation $\mathcal{G}, \varphi \models_S C$, witnessing that the constraint C is satisfied relative to the model $\mathcal{G}, \varphi$ with *support* S. The notion of support is used to distribute ownership of scopes in the graph in a *disjoint* fashion over the constraint. The notion of unique ownership over scopes in the graph gives Statix constraints a separation logic flavor, which is visible in the satisfaction rules for the ∇t constraint and conjunction $C_1 \wedge C_2$. The intuition we gave for ∇ can be captured formally as $\nabla x \wedge \nabla y \wedge (x = y) \equiv \bot$. The rule for conjunction *separates* the support S into two *disjoint* parts ($S = S_1 \sqcup S_2$) and distributes this among the left and right operands. Entailment and equivalence of constraints are defined as usual:

$$C_1 \Vdash C_2 \triangleq \forall\ \mathcal{G}\ \varphi\ S.(\mathcal{G}, \varphi \models_S C_1 \implies \mathcal{G}, \varphi \models_S C_2) \quad C_1 \equiv C_2 \triangleq (C_1 \Vdash C_2) \wedge (C_2 \Vdash C_1)$$

We emphasize that due to the presence of scope ownership, Statix constraints do not enjoy all the equivalences that, for example, ML-constraints do (Pottier and Rémy, 2005). A general rule $(C_1 \Vdash C_2) \implies (C_1 \equiv C_1 \wedge C_2)$ does not hold, since both C_1 and C_2 may require ownership over the same scopes. This is consistent with the rules for separating conjunction in affine separation logics.

3.4 Executing Statix Specifications

In this section we discuss how Statix specifications can be executed as type checkers.

3.4.1 Constraint Solving by Simplification

The requirement to use Statix specifications as executable type checkers has guided its design. In particular, the following three concerns have been important: 1. Specifications should have a declarative meaning that is independent of the operational interpretation. 2. Users should not be concerned with execution order when writing specifications. 3. The implementation should not rely on expensive techniques such as full back-tracking, which make it difficult to reason about performance. To achieve this, we take a constraint solving approach. User-defined constraints are simplified, using the rules from the specification, to built-in constraints, which are solved using algorithms for unification and name resolution. By disallowing overlapping guards, rule selection follows a committed choice strategy, that is, no backtracking is needed. We use the example in Fig. 3.15 with Statix rules for a recursive let construct to illustrate issues around order and soundness.

Constraint Simplification The solver maintains a state consisting of a set of constraints to solve, a unifier, and a scope graph. The ultimate goal is to eliminate all constraints. The resulting unifier and scope graph are the solution. We illustrate the simplification process by discussing the first steps of checking the example program in Fig. 3.15. We start with a single constraint:

$$\mathsf{typeOfExp}(\#0,\mathtt{\textbf{letrec}\ odd_1\ =\ ...;\ even_3\ =\ ...\ \textbf{in}\ ...,T})$$

We assume a scope graph with a single scope #0. The constraint is simplified using the first rule in Fig. 3.15, resulting in the constraints

$$\nabla s_{\mathrm{b0}} \wedge s_{\mathrm{b0}} \xrightarrow{\mathsf{P}} s \wedge \mathsf{bindOK}(s_{\mathrm{b0}},\ldots) \wedge \mathsf{typeOfExp}(s_{\mathrm{b0}},\ldots,\mathtt{T})$$

A fresh unification variable s_{b0} is created for the locally quantified variable s_{b}. Solving ∇s_{b0} results in a fresh scope #1, which is added to the scope graph, as well as a substitution $s_{\mathrm{b0}} \mapsto \#1$ in the unifier. The edge constraint $s_{\mathrm{b0}} \xrightarrow{\mathsf{P}} s$ is solved by adding an edge to the scope graph from scope #1 to scope #0. Note the order: the edge could only be added after the fresh scope was created. Next, the solver needs to ensure that the constraint $s \xrightarrow{:} (\mathtt{x},\mathtt{T})$ from the `bindOK` rule is solved for both binds, before attempting to resolve the references in the expressions. How the solver ensures that this is the case is the subject of most of the rest of this section.

Delayed Constraints In general, the solver randomly selects the next constraint to solve from the constraint set. A satisfied constraint results in an updated unifier and scope graph, and the constraint is removed from the constraint set. If a user-defined constraint is simplified, the constraints from the applied rule body are added to the constraint set. However, sometimes a constraint cannot be solved yet. For example, a *guard* constraint $\mathtt{T} = \mathtt{\textbf{num}}$ cannot be discharged if `T` is a unification variable

$$\mathsf{typeOfExp}(s, \texttt{letrec}\ \bar{\mathtt{b}}\ \texttt{in}\ \mathtt{e}, \mathtt{T}) \leftarrow \exists s_{\mathrm{b}}.\ \nabla s_{\mathrm{b}} \wedge s_{\mathrm{b}} \xrightarrow{\mathtt{P}} s \wedge \mathsf{bindOK}(s_{\mathrm{b}}, \bar{\mathtt{b}}) \wedge \mathsf{typeOfExp}(s_{\mathrm{b}}, \mathtt{e}, \mathtt{T})$$

$$\mathsf{bindOK}(s, \mathtt{x}_i\ \mathtt{=}\ \mathtt{e}) \leftarrow \exists \mathtt{T}.\ \mathsf{typeOfExp}(s, \mathtt{e}, \mathtt{T}) \wedge s \xrightarrow{:} \blacksquare\ (\mathtt{x}_i, \mathtt{T})$$

```
letrec odd1 = fun(n2:num){ ... even3(n4-1) ... }
       even5 = fun(n6:num){ ... odd7(n8-1) ... } in odd9 11
```

Figure 3.15: Statix rules for a recursive let extension of the simply-typed lambda calculus rules of Fig. 3.12 and an example program using the letrec construct.

without a substitution. In this case, the constraint is delayed, and put back into the constraint set. Other constraints may instantiate T, after which the equality can be tested. Just unifying T would not be sound in general due to the committed choice strategy. Constraint solving continues until all constraints are resolved or remain delayed. Similar techniques are found in other constraint solvers that support guarded rules and constraints, such as CHR (T. Frühwirth and Brisset, 1995).

3.4.2 Name Resolution Algorithm

The calculus presented in Section 3.2.1 gives a precise definition of name resolution. In this section we discuss the name resolution algorithm that is used in the implementation of Statix. The algorithm essentially implements an ordered depth-first search in the scope graph. The well-formedness predicate *WFL* is used to control depth, and the label order $<_l$ is used to control breadth and cut-off of the search. Cyclic paths are also disallowed (by the condition on rule (NR-Cons) in Fig. 3.1), so the algorithm is terminating.

We show how the algorithm operates using the example scope graph in Fig. 3.16. The parameters we show are for resolving the reference $\mathtt{x}_3$. Edges have labels P and Q. Path well-formedness *WFL* states that well-formed paths cannot follow Q edges after P edges. Data well-formedness *WFD* matches declarations with the same name x as the reference. The label order $<_l$ prefers Q over P, and local declarations over both. Finally, the data order $\leq_{\mathrm{T}}$ states that declarations via more specific paths always shadow declarations reachable via less specific paths. Resolution starts in scope #3.

The dashed lines show the order in which the algorithm traverses the graph. The search starts at scope #3 of the reference ①, and tries the different labels in order. The most specific label is $, for the local declarations. This scope has no local declarations, so the Q edge to #4 is traversed ②. Again the algorithm starts with the most specific label. In this case there are local declarations, and the path up to here is well-formed. The local declaration $\mathtt{y}_2$ is matched ③, but the match fails, so the search continues. There are no Q edges in scope #4, so P edges are followed next. According to path well-formedness, no more P steps are allowed. Therefore the search

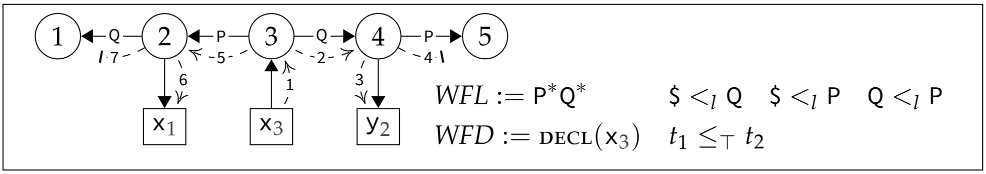

1
Q
2
P
3
Q
4
P
5
7
5
2
4
6
1
3
x1
x3
y2
$WFL := \mathsf{P}^*\mathsf{Q}^*$
$\$ <_l \mathsf{Q}$ $\$ <_l \mathsf{P}$ $\mathsf{Q} <_l \mathsf{P}$
$WFD := \textsc{decl}(\mathsf{x}_3)$
$t_1 \leq_\top t_2$

Figure 3.16: Name resolution example. The dashed arrows visualize the search in the graph, and the label numbers indicate the search order.

is cut-off ④, and continues at scope #3. After trying local declarations and Q edges, the P edge is followed ⑤, which is allowed at this point, because we are back at the starting scope and the path is empty. The local declarations contain the matching declaration x_1, which is added to the results ⑥. Next, the algorithm could continue following Q edges. But according to the data order, the result found so far shadows any results reachable via the less specific paths that are tried next, so the search is cut off ⑦, and the algorithm returns $\{(\#3 \cdot \mathsf{P} \cdot \#2, \mathsf{x}_1)\}$ as the final result. Note that the well-formedness cut-off can always be performed, but the order cut-off only when the data order is always true.

3.4.3 Sound Name Resolution

The declarative semantics of query constraints is expressed in terms of resolution in a complete scope graph. However, during constraint solving, we gradually build the scope graph. Invoking the resolution algorithm on an intermediate, incomplete graph may yield a different result than invoking it on the final graph. This is potentially unsound, and should therefore be prevented. We explain the problem and our solution using the example from Fig. 3.15.

Problem Fig. 3.17 shows some of the constraints that occur when checking our example, as well as the relevant fragment of the final scope graph. Scope #1 is the let scope, while scopes #2 and #3 are the function scopes. We only show the references and declarations for the let bindings. We discuss the constraints for binds, on lines 1 and 4, and the declaration and query they eventually simplify to, on lines 2 and 3, and 5 and 6, respectively. We refer to the constraints by their line numbers.

Consider a state where 1 and 4 are simplified, but 2, 3, 5 and 6 remain. Intuitively, we see that 5 must be solved before 3, and 2 before 6. This conclusion requires detailed knowledge of where declarations are added, and what names they have. An easy approach is to delay all queries until the graph is fully known. This solution is sound, because all queries are performed on the final graph. However, this solution is also unsatisfactory, because the solver would not be able to solve type dependent names, such as field accesses. Our goal is a solution that is sound, but does allow interleaved building and querying of the scope graph.

$$\texttt{bindOK}(\#1,\texttt{odd}_1\ \texttt{=}\ \texttt{...}\ \texttt{even}_3\ \texttt{...}) \tag{3.1}$$

$$\rightarrow\ \texttt{...}\ \#1 \overset{:}{\longrightarrow}\!\blacksquare (\texttt{odd}_1\texttt{,FUN(T1, U1)})\ \texttt{...} \tag{3.2}$$

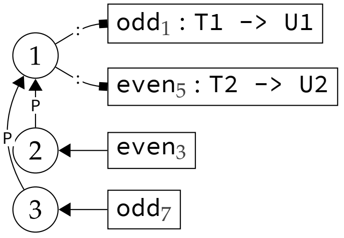
$$\texttt{...}\ \texttt{even}_3 \text{ in } \#2 \overset{:}{\mapsto} (\texttt{d1,S1})\ \texttt{...} \tag{3.3}$$

$$\texttt{bindOK}(\#1,\texttt{even}_5\ \texttt{=}\ \texttt{...}\ \texttt{odd}_7\ \texttt{...}) \tag{3.4}$$

$$\rightarrow\ \texttt{...}\ \#1 \overset{:}{\longrightarrow}\!\blacksquare (\texttt{even}_5\texttt{,FUN(T2, U2)})\ \texttt{...} \tag{3.5}$$

$$\texttt{...}\ \texttt{odd}_7 \text{ in } \#3 \overset{:}{\mapsto} (\texttt{d2,S2})\ \texttt{...} \tag{3.6}$$

(a) Bind constraints and part of their simplification



(b) Fragment of the final scope graph

Figure 3.17: Partial constraints and scope graph for the letrec example program.

Tracking Possible Scopes Extensions in the Constraint Set Our solution consists of two parts. First, the solver tracks possible scope extensions in the constraint set. Second, the resolution algorithm aborts when it searches a scope that may be incomplete. We first consider the scenario where constraints 2, 3, 5, and 6 are in the constraint set. Constraints 2 and 5 both extend scope #1 in the $\overset{:}{\longrightarrow}\!\blacksquare$ relation. If the solver tried to solve constraint 6, the resolution algorithm would search scope #3, then step to scope #1, where it would try to find local data for the $\overset{:}{\longrightarrow}\!\blacksquare$ relation. However, since there are constraints in the constraint set that extend scope #1 in that relation, resolution is aborted, and the query constraint delayed. This scheme forces constraints 2 and 5 to be solved before 3 and 6. However, possible incompleteness in other scopes, for example the parent of scope #1, would not block solving these constraints.

The situation is more complicated when we consider user-defined constraints. For example, if the solver is in a state where constraint 1 is simplified to 2 and 3, but 4 is not simplified yet. Solving constraint 3 next would be unsound, because the declaration even$_5$ is still missing in the scope graph. The fact that the rule for `bindOK` contains a $s \overset{:}{\longrightarrow}\!\blacksquare \ldots$ constraint, allows us to conclude that scope #1, the first argument to `bindOK`, may be extended in the $\overset{:}{\longrightarrow}\!\blacksquare$ relation. However, the situation is not always that simple. Consider the following three rules for a predicate `c`, all extending a scope:

$$\begin{aligned}
\texttt{c}(\ldots,s,\ldots) &\leftarrow \cdots \wedge s \overset{:}{\longrightarrow}\!\blacksquare \ldots \wedge \ldots \\
\texttt{c}(\ldots) &\leftarrow \cdots \wedge \nabla s \wedge s \overset{:}{\longrightarrow}\!\blacksquare \ldots \wedge \ldots \\
\texttt{c}(\ldots) &\leftarrow \cdots \wedge \texttt{d}(\ldots,s,\ldots) \wedge s \overset{:}{\longrightarrow}\!\blacksquare \ldots \wedge \ldots
\end{aligned}$$

The first rule is the case of `bindOK`, where the extended scope is passed as an argument. Here we know how `c` might extend the scope. In the second rule, the scope that is extended is locally fresh. Because the scope is fresh, we know that the extension does not concern any of the scopes already in the scope graph. In the third rule, the scope is only restricted by a predicate `d`. Determining the potential scope value(s) may be impossible, or require a sophisticated data-flow analysis. To keep things simple, Statix does not allow rules of the third form. This restriction is reasonable if the scope graph is seen as a rich environment. Although lookups originate in many places, construction is inherently local. Computing the possible scope extensions for all constraints is now a simple data flow analysis on a Statix specification. This static information is used to track possible scope extensions for user-defined constraints.

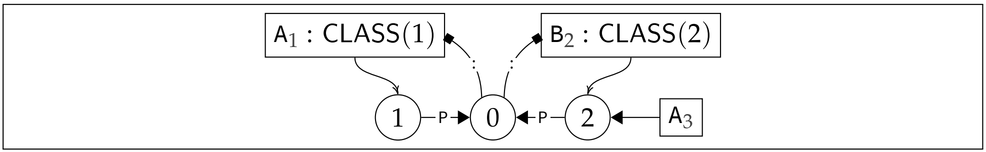
So far all scopes in the constraint set were known. In a situation where the constraint set contains a constraint `s1` $\overset{:}{-\!\!-\!\!\blacksquare}$ ..., where `s1` is a free unification variable, we cannot be sure which scope can be extended. Therefore, as long as such a constraint is present, all scopes are marked incomplete for the $\overset{:}{-\!\!-\!\!\blacksquare}$ relation. However, our rule restrictions ensure that these scope variables are eventually substituted by constraint arguments or fresh scopes.

3.4.4 Incompleteness

The technique presented above ensures that if a query is resolved, its result will not be invalidated when the scope graph is extended. Therefore, soundness with respect to the declarative semantics is achieved. However, this method over-approximates possible extensions, and does not take the values stored in relations into account. Therefore, it is incomplete, and it is possible to define constraints that get stuck, because the solver cannot find an order it knows to be safe.

In general, constraints get stuck if the extension of a scope depends on the resolution of a query via that scope. To illustrate this, we use an example from FJ, shown in Fig. 3.18. It shows an incomplete scope graph, which is the result of checking two classes, `A` and `B extends A`. Sub-classing is modeled by an `S` edge from sub-class to super-class. Scope #1 of the super-class is discovered by resolving $\mathtt{A}_3$ in scope #2, after which the `S` edge is added. To solve these constraints, the resolution of $\mathtt{A}_3$ must be allowed while scope #2 is incomplete in `S`.

Whether resolution is possible depends on the resolution parameters. The table in Fig. 3.18 lists the possible values for well-formedness, label order, and data order for this scenario. The options are the allowed first step of the path, and the relative specificity of labels `P` and `S`. First row: if the first step can only be a `P` step, the incompleteness in `S` is irrelevant, and the reference can be resolved. Second row: if the first step can only be a `S` step, then the declarations are unreachable, and the constraints cannot be solved. Third row: if both steps are well-formed, the label order is relevant. If `P` is more specific than `S`, the reference can be resolved, provided the data order agrees.


$\mathsf{A}_1 : \mathsf{CLASS}(1)$
$\mathsf{B}_2 : \mathsf{CLASS}(2)$
:
:
1
P
0
P
2
A_3

$\mathsf{A}_3\ \mathsf{in}\ \#2 \mapsto (d, \mathsf{CLASS}(s)) \qquad \#2 \xrightarrow{\mathsf{S}} s$

WFL	$\mathsf{P} <_l \mathsf{S}$	$\mathsf{P} \not<_l \mathsf{S}$
$\mathsf{P}.p$	Solved	Solved
$\mathsf{S}.p$	Stuck	Stuck
$(\mathsf{P}\vert\mathsf{S}).p$	Solved if $\leq_d$ is always true	Stuck

Figure 3.18: Scope graph, constraints, and stuckness for the different possible label well-formedness *WFL*, label order $<_l$, and data order $\leq_d$ parameters

3.5 Evaluation

We have evaluated the expressiveness of the scopes-as-types approach and the Statix language by means of several case studies implemented with the Spoofax Language Workbench (Kats and Visser, 2010) extended with Statix.

Statix Implementation In this paper we have defined a Statix core language with mathematical notation and a formal declarative semantics. We have also developed a full fledged Statix programming language with a concrete ASCII notation embodying the design described in this paper. We have implemented the Statix language itself with Spoofax. The implementation consists of a syntax definition in SDF3, a type checker in NaBL2 (the precursor of Statix), and a solver in Java. The syntax definition provides an Eclipse editor with syntax checking and highlighting of Statix definitions. The NaBL2 definition provides type checking and reference resolution (jump to declaration) for Statix definitions in the editor. The Statix solver interprets the AST of a Statix specification applied to the AST of an object language program. Currently, the solver only accepts or rejects an object language program. It does not yet give error messages to explain a failure. The solver is integrated with object language editors and in the SPT testing framework (Kats et al., 2011). The Statix language and solver are available in the current continuous releases of Spoofax.

Case Studies To validate the expressiveness of Statix and the operation of the solver, we have developed Statix specifications for several model languages: STLC-REC, System F, and FGJ. These languages represent points in the design space of type systems and are well-known benchmarks to validate approaches and tools for type systems. The languages are *models* in the sense that they are reduced to the essence of a feature in type systems and abstract from details irrelevant to that feature. Thus, it becomes easier to appreciate the key ideas of the encoding. While such case studies do not demonstrate that the approach scales to specification of full-fledged languages (and the feature interaction that comes with those), they do provide evidence of the expressiveness of the approach. For each language we have defined a Spoofax project including a syntax definition in SDF3, a static semantics definition in Statix, example programs, and an SPT test suite to test the static semantics. The implementations provide source code editors with syntax highlighting and type checking, and automated execution of the test suites.

Validation We have validated the Statix definitions in two ways. First, for each language we have constructed test suites with key examples testing the corner cases of the language. Second, we have constructed the definitions by closely following existing formalizations, replacing traditional name binding mechanisms (environments, association lists, class tables) with the corresponding scope graph mechanisms. In Section 3.2 we have discussed in detail type system specifications of STLC, SLTC-REC, FJ, and System F. Those specifications are 'backports' from the Statix specifications in our case studies to traditional inference rules using scope graphs for name resolution. (The specification of FGJ combines the ideas of FJ and System F.) These presentations do not suffer from the 'clutter' that comes from an encoding in ASCII, but instead use the notational abstraction of mathematics (reducing judgements to turnstiles, vector notation for lists, concrete syntax instead of term syntax, etc.), which should make it easier for the reader to appreciate the commonalities and differences with standard formalizations. In Section 3.2 we have discussed how the specifications for the case studies compare to standard formalizations of type systems.

Artifact The Statix specifications from the case studies are available in the artifact accompanying the paper, which is publicly available at https://github.com/MetaBorgCube/oopsla18-artifact. The prototype implementation of Statix has been integrated in the Spoofax Language Workbench.

3.6 Related Work

In this section we discuss how our approach to type system specification compares to other approaches, focusing on the support for name binding and executability of specifications as type checkers. In Section 3.2 we compared the scope graph approach with representations for name binding in traditional type system definitions.

Name Binding Languages Name binding is a concern in all kinds of language engineering processes, and that can benefit from more specialized support. When for-

```
t ::=                                       p ::=
  | x                                         | x          b = x
  | (t1, t2)                                  | (p1, p2)   b = b(p1) U b(p2)
  | let p = t in t'   bind b(p) in t'         | p1 | p2    b = b(p1) U b(p2)
```

Figure 3.19: Pattern binders in Ott (from Sewell et al., 2010)

$$
\begin{aligned}
\mathsf{typeOfExp}(s,\mathtt{x}_i,\mathtt{T}) &\leftarrow \exists \mathtt{x}_j.\,\exists \mathtt{T'}.\,\mathtt{x}_i \text{ in } s \overset{\cdot}{\mapsto} (\mathtt{x}_j,\mathtt{T'}) \wedge \mathsf{eqTypes}(\mathtt{T'},\mathtt{T})\\
\mathsf{typeOfExp}(s,(\mathtt{t}_1,\ \mathtt{t}_2),\mathtt{T}_1 \times \mathtt{T}_2) &\leftarrow \mathsf{typeOfExp}(s,\mathtt{t}_1,\mathtt{T}_1) \wedge \mathsf{typeOfExp}(s,\mathtt{t}_2,\mathtt{T}_2)\\
\mathsf{typeOfExp}(s,\mathbf{let}\ \mathtt{p}\ =\ \mathtt{t}_1\ \mathbf{in}\ \mathtt{t}_2,\mathtt{T}_2) &\leftarrow \exists \mathtt{T}_1.\,\exists s_l.\,\nabla s_l \wedge s_l \xrightarrow{\mathsf{P}} s \wedge \mathsf{typeOfExp}(s,\mathtt{t}_1,\mathtt{T}_1)\\
&\qquad \wedge \mathsf{typePat}(s_l,\mathtt{p},\mathtt{T}_1) \wedge \mathsf{typeOfExp}(s_l,\mathtt{t}_2,\mathtt{T}_2)\\
\mathsf{typePat}(s,\mathtt{x}_i,\mathtt{T}) &\leftarrow s \overset{:}{\longrightarrow}\!\blacksquare\ (\mathtt{x}_i,\mathtt{T})\\
\mathsf{typePat}(s,(\mathtt{p}_1,\ \mathtt{p}_2),\mathtt{T}_1 \times \mathtt{T}_2) &\leftarrow \mathsf{typePat}(s,\mathtt{p}_1,\mathtt{T}_1) \wedge \mathsf{typePat}(s,\mathtt{p}_2,\mathtt{T}_2)\\
\mathsf{typePat}(s,(\mathtt{p}_1\ \mid\ \mathtt{p}_2),\mathtt{T}) &\leftarrow \exists s_1.\,\exists s_2.\,\nabla s_1 \wedge \nabla s_2 \wedge s \xrightarrow{\mathsf{O}} s_1 \wedge s \xrightarrow{\mathsf{O}} s_2\\
&\qquad \wedge \mathsf{typePat}(s_1,\mathtt{p}_1,\mathtt{T}) \wedge \mathsf{typePat}(s_2,\mathtt{p}_2,\mathtt{T})\\
&\qquad \wedge \mathsf{comp}(s_1,s_2)
\end{aligned}
$$

Figure 3.20: Pattern binders and types in Statix

malizing a language for use in mechanically verified meta-theory, details of name binding (e.g., substitution functions) are important, but tedious to define. Various libraries and DSLs have been developed to automate the support for name binding in proof assistants such as Coq. For example, AutoSubst (S. Schäfer et al., 2015) is a Coq library that derives substitution and renaming functions and lemmas about their properties from annotations on an inductive type definition; Ott (Sewell et al., 2010) is a DSL to define type systems and reduction rules for languages with name binding, from which it automatically generates data types and substitution functions for different proof assistant back-ends; Lem (Mulligan et al., 2014) and Needle & Knot (Keuchel et al., 2016) provide similar support. These tools follow similar schemas to define bindings: an annotation in the constructor signature indicates that a binding occurrence is bound in one or more sub-terms of the binding construct.

Fig. 3.19 shows the definition of OCaml patterns in Ott, a non-trivial example, since the variables occurring (deep) inside the pattern `p` are bound in the body `t'` of the let. To realize this, an auxiliary function `b(p)` is defined (alongside the inductive definition) that collects the binding occurrences. The typing rules for such a language are defined separately, and assert that the sub-patterns of the or-pattern declare the same variables. Fig. 3.20 shows the definition in Statix of the `typeOfExp` and `typeOfPat` predicates for the same language. The predicates define the binding *and* typing of expressions and patterns without a separate collection traversal. Pattern variable binders are added as declarations to the scope graph (4th rule) with a constraint variable `T` as type. Unification with the requirements from the context and any references will specialize the type assignment. Checking that the branches of an or-pattern should define the same variables is modeled by creating a new scope for each branch, and checking with the comp predicate that they declare the same variables with the same types (similar to structural record comparison in Section 3.2.3).

The Ott definition for Lightweight Java (Strnisa and Parkinson, 2011) follows the F(G)J formalization of Igarashi et al. (2001) by defining projections on syntactic entities to look-up information instead of using binding specifications. By contrast, our definition of F(G)J uses scope graphs for such non-lexical bindings as well (Section 3.2.4).

Statix does not (yet) support generating infrastructure for proof assistants, although scope graph libraries exist for use in type safety proofs in both Coq (Bach Poulsen et al., 2016) and Agda (Bach Poulsen et al., 2018). We intend to extend these libraries to the scopes-as-types extension presented in this paper and connect Statix type checkers to intrinsically-typed definitional interpreters.

Abstractions for Type Checker Implementation The PLT Redex semantic specification framework (Felleisen et al., 2009) can also be used to define type systems similar in style to traditional inference rules using type environments. Redex supports the definition of lexical binding forms as part of language definitions, which cause all uses of the term to avoid capture (Stansifer, 2016). Its name binding DSL can express only simple forms of name binding. Redex rules can be used as a random generator of (well-typed) terms (Fetscher et al., 2015; Lampropoulos et al., 2017), an interpretation we intend to explore for Statix in future work. The Turnstile language (Chang et al., 2017), which shares our goal of bridging specification and implementation, allows writing a type judgement-like syntax, which directly corresponds to a macro-based type checker implementation that reuses the binding mechanism (Flatt, 2016) of the underlying macro system.

K is a framework aimed at the executable definition of dynamic semantics based on rewriting (Rosu and Serbanuta, 2010). The context of evaluation (environment, store) is represented by (nested) *configurations*, which can be accessed using powerful pattern matching expressions. K can also be used to define static semantics as a set of rules that rewrite a program to a type in abstract interpretation style (Ellison III, 2008). In that setting, configurations are used to represent the typing context. No further abstractions are provided for the treatment of name binding.

The JastAdd attribute grammar system (Ekman and Hedin, 2006, 2007a; Hedin, 2009) uses reference attributes to link references to their declarations in the abstract syntax tree. Name resolution is defined using inherited parameterized attributes that search for a declaration node. For example, the definition of the resolution of a field in a class in Fig. 3.21 first considers the member fields of a class, and if none is found, it recursively continues the search in the superclass. This is a programmatic

```
syn FieldDecl ClassDecl.memberField(String name) {
  for(int i = 0; i < getNumBodyDecl(); i++)
    if(getBodyDecl(i).isField(name))
      return (FieldDecl)getBodyDecl(i);
  if(getSuper().type().memberField(name) != null)
    return getSuper().type().memberField(name);
  return null;
}
```

Figure 3.21: Field resolution in a class and its superclasses in JastAdd (from Ekman and Hedin, 2006).

encoding of the name resolution query in rule (FJ-Field) in Fig. 3.8. Scope graphs provide a reusable abstraction for characterizing such name resolution strategies.

Constraint-Based Approaches The design of Statix was inspired by *Constraint Handling Rules* (CHR; T. Frühwirth, 2009), which have been used to define type checkers. CHR provides 'simpagation' rules that can match multiple constraints in the constraint set simultaneously, and thus extend or reduce the constraint set. Statix only provides rules that consider a single constraint, to ensure deterministic execution. The built-in theory of name resolution replaces the constraint store to support the context-sensitive nature of type checking.

Specifying and implementing type checkers using constraints is an established technique (e.g., Odersky et al., 1999; Pottier and Rémy, 2005; Simonet and Pottier, 2007; Sulzmann and Stuckey, 2008; Vytiniotis et al., 2011). Usually name resolution is considered part of the constraint *generation* phase. When it is part of the constraint language (e.g., Pottier and Rémy, 2005), the constraints mimic the (lexical) binding structure from the object language, which is a limitation for the definition of languages with type-dependent name resolution. To extend Hindley-Milner type inference (Damas and Milner, 1982; Hindley, 1969; Milner, 1978) to System F, Pierce and Turner (2000) and Odersky et al. (2001) developed *bidirectional type checking*, which carefully controls the introduction of constraint variables for the inference of type parameters. Our definition of System F (Section 3.2.5) does not include inference of type parameters, but the bidirectional approach seems to be applicable.

3.7 Conclusion

We have demonstrated that the scope graph framework covers a wide range of name binding patterns in programming languages, including structural and parameterized types. We have also presented the design of Statix, a language for the specification of type checkers that uses scope graphs to abstract from the stratification of collecting and using binding information. We believe that the scope graph approach has the potential for standardizing the treatment of name binding in programming languages and their tools, just as context-free grammars have done for syntax.

Acknowledgments We thank Reuben Rowe, Robbert Krebbers, Andrew Tolmach, Robby Findler, Simon Peyton Jones, and the OOPSLA reviewers for their comments on earlier versions of this paper. This research was funded by the NWO VICI *Language Designer's Workbench* project (639.023.206).

II

Interpretation

Knowing When to Ask

Abstract There is a large gap between the specification of type systems and the implementation of their type checkers, which impedes reasoning about the soundness of the type checker with respect to the specification. A vision to close this gap is to automatically obtain type checkers from declarative programming language specifications. This moves the burden of proving correctness from a case-by-case basis for concrete languages to a single correctness proof for the specification language. This vision is obstructed by an aspect common to all programming languages: name resolution. Naming and scoping are pervasive and complex aspects of the static semantics of programming languages. Implementations of type checkers for languages with name binding features such as modules, imports, classes, and inheritance interleave collection of binding information (i.e., declarations, scoping structure, and imports) and querying that information. This requires scheduling those two aspects in such a way that query answers are stable—i.e., they are computed only after all relevant binding structure has been collected. Type checkers for concrete languages accomplish stability using language-specific knowledge about the type system.

In this paper we give a language-independent characterization of necessary and sufficient conditions to guarantee stability of name and type queries during type checking in terms of *critical edges in an incomplete scope graph*. We use critical edges to give a formal small-step operational semantics to a declarative specification language for type systems, that achieves soundness by delaying queries that may depend on missing information. This yields type checkers for the specified languages that are sound by construction—i.e., they schedule queries so that the answers are stable, and only accept programs that are name- and type-correct according to the declarative language specification. We implement this approach, and evaluate it against specifications of a small module and record language, as well as subsets of Java and Scala.

4.1 Introduction

In an ideal world, programming language designers should not have to deal with accidental complexity when defining and implementing languages. Some aspects of language design are already close to realizing this ideal. For example, parser generators make it possible to obtain parsers from declarative grammar specifications, thus abstracting over the accidental complexity of implementing parsing. There should

Published as Arjen Rouvoet, Hendrik van Antwerpen, Casper Bach Poulsen, Robbert Krebbers, and Eelco Visser (2020a). "Knowing when to ask: sound scheduling of name resolution in type checkers derived from declarative specifications." In: *Proceedings of the ACM on Programming Languages* OOPSLA. DOI: 10.1145/3428248. Copyright © 2020 Owner/Author. Licensed under a Creative Commons Attribution International 4.0 License.

be similar support for generating implementations of type checkers from declarative specifications of type systems.

The variety of language features found in real-world languages presents many challenges in the way of this ideal. This paper focuses on the challenges presented by name resolution, an aspect common to all programming languages. Many language features found in actual languages interact with name resolution. Modules, imports, classes, interfaces, inheritance, overloading, and type-dependent member access to objects and records are a few examples that are commonplace. Implementing type checkers for languages with such features is complicated because the use of names in programs causes dependencies between type-checking tasks, and requires that the *construction* of symbol tables and type environments is interleaved with *querying* those data structures. Evaluating a query too early may result in an unstable answer—i.e., an answer that is invalidated by subsequent additions to the environment or symbol table. A wrong answer can have far reaching consequences, either compromising the soundness of the type checker, or later requiring backtracking on an arbitrary amount of work that depends on the wrong answer.

Consider, for example, the valid Scala program in Fig. 4.1a. A type checker working its way forward through the program would initially resolve `import B._` to the imported object `M.B`, and type check the remainder of the body of `A` under the resulting environment. If only then it encounters the local declaration of `B` on line 7, it needs to redo the type checking of the body of `A` because the local definition shadows the earlier imported declaration.

To avoid this, the interleaving chosen by the type checker must ensure that *query resolution is stable*—i.e., that answers to queries that consult the symbol table are not invalidated by subsequent additions to the environment or symbol table. This can be a non-trivial scheduling problem because environment and symbol table construction can also *depend* on answering queries.

Languages often have many features that interact with name binding and disambiguation, and as a consequence it can be difficult to construct schedules that guarantee query stability. The simple valid Scala program in Fig. 4.1b shows for example how classes and inheritance interact with name resolution. In this program, the `f` on line 2 resolves to the `def f` on line 6; but for this resolution to succeed, the qualified reference `B.D` on line 1 must first have been resolved to the `D` on line 4, to make the bindings in class `class D` reachable from the body of `class A`. Resolving `B.D` in turn depends on: (1) resolving the `B` in `B.D` to `object B` on line 4; and (2) resolving the `C` in the extends clause for `object B` on line 4 to the `C` declaration on line 5.

Determining these dependencies requires a good understanding of the binding and disambiguation rules of a language. The type checking algorithm must take all these dependencies into account, so that names are only resolved once all information that is relevant to their resolution is collected. If this is the case, then the result

```
object M { object B { ... }}
import M.B;
object A {
  import B._;
  ...
}
object B { ... }
```

(a) Forward reference to shadowing definition.

```
class A extends B.D {
  def g:Int = f
}
object B extends C {}
class C {
  class D { def f:Int = 1 }
}
```

(b) Inheritance in Scala.

Figure 4.1: Scala examples.

of name resolution is stable. Type checker implementations use various strategies for stratifying or scheduling the collection and querying of name binding information. Every type checker must, implicitly or explicitly, solve this scheduling problem. For example, Haskell's binding restrictions ensure that binding collection and resolution can be separated into static passes over the program, whereas Scala and Rust require type-dependent name resolution, which requires interleaving type checking and name resolution. A key property of sound strategies is that names are only resolved after all the relevant information has been collected.

The concrete strategies are irrelevant for understanding and reasoning about the underlying type system, but crucial to a correct implementation of the type checker. This tension between implementation and specification is felt by language designers. For example, the Rust language developers write the following about specifying name binding in the language:[1]

> *Whilst name resolution is sometimes considered a simple part of the compiler, there are some details in Rust which make it tricky to properly specify and implement.*

And in reply to changes to the design and implementation of name binding, a contributor states:[2]

> *I'm finding it hard to reason about the precise model proposed here, I admit. I wonder if there is a way to make the write up a bit more declarative.*

A more declarative specification should allow reasoning about name binding without having to rely on an understanding of the operational details such as the scheduling of name and type queries. But if we want to obtain type checkers from these

[1] https://github.com/nrc/rfcs/blob/name-resolution/text/0000-name-resolution.md

[2] https://github.com/rust-lang/rfcs/pull/1560

declarative specifications, we need to be able to automatically construct sound schedules. In this paper we give a language independent explanation of necessary and sufficient conditions to guarantee stability of name and type queries during type checking. We use this to make declarative type system specifications executable as type checkers for the specified language. Using this approach, we can guarantee that the resulting type checkers are *sound* with respect to the formal declarative semantics of the specifications, as well as *confluent*. These important properties of type checkers are proven once-and-for-all for languages specified using our formalism, rather than on a language-by-language basis.

Problem We start from a specification of an object language's static semantics in the meta-language Statix (van Antwerpen et al., 2018). Language specifications in Statix are given by typing rules, written as predicates on terms, types, and a *scope graph* (Néron et al., 2015). Scope graphs generalize language specific notions of type environments and symbol tables. A distinguishing feature of Statix are its *scope graph assertions and queries*, which can be used to give high-level specifications of name resolution. These assertions can express fine-grained name resolution rules, which enable high-level specification of, for example, shadowing rules of Java and Scala.

The problem we face is to derive a type-checker from a Statix specification. Statix's scope graph assertions and queries make it possible to give high-level specifications of name binding, but, at the same time, make the problem of deriving these type checkers more difficult. In particular, we have to solve a generalized version of the scheduling problem described above. That is, we need a *general* characterization of the conditions under which it is sound to query symbol tables and type environments during type checking. We then need to derive a type checker from a Statix specification in such a way that these conditions are always satisfied.

The general approach to deriving type checkers from Statix specifications is already sketched by van Antwerpen et al. (2018), who provide a Java implementation. They explain the problem with unsound name resolution when queries answers are unstable, and they claim that their implementation implements a sound strategy. This strategy, however, is only informally described, and lacks evidence of its soundness.

This paper addresses both those deficiencies by formalizing the derivation of type checkers from Statix specifications, and proving soundness. Our formalization of the operational aspects revealed that the Java implementation of Statix is, in fact, not confluent Rouvoet, van Antwerpen, et al., 2020c, Appendix A, which we address in this paper by refining the scope graph primitives. Confluence is an important property because Statix implements a non-deterministic solver. It ensures that the solver does not have to backtrack on evaluation order. In order to formalize the soundness and confluence results, we develop a theory around the novel concept of *critical edges in scope graphs*. We believe that this concept is a useful device in both the design of languages, and the implementation of their type checkers. We also hope that the formalization of the operational semantics of Statix makes it feasible to port the novel ideas of Statix about the high-level specification of name binding to other formalisms and type checker implementations.

Approach To enable this formalization, we first introduce Statix-core. This core language refines and simplifies the previous formulation of the Statix meta-language. The declarative semantics of Statix-core is similar to the declarative semantics of Statix, and explains what are valid type derivations of a specified language. In other words, it explains when a *given* object-language program, together with a type assignment and a scope graph model of its binding, *satisfies* the specified static semantics of an object language.

We equip this refined core of Statix with a novel small-step operational semantics. This operational semantics takes a specification and an object-language program, and then computes a type assignment and a scope graph, thus fulfilling the task of a type checker for the object language. The key question of this paper arises when we try to define how queries in Statix compute. What are the conditions that ensure that the answer to a scope graph query is stable under future additions to the scope graph model of binding in the program? Or, how do we *know when to ask* a query?

To make the condition for query answering precise, we introduce the new idea of *critical edges* for a query in a scope graph extension, precisely characterizing missing dependencies of the query. Conceptually, query answers that are computed in a partial scope graph are stable if recomputing the answer in a complete model of the program yields the same result. We will show that it is safe to answer a query in a partial graph $\mathcal{G}$ when the complete model contains no critical edges for the query with respect to $\mathcal{G}$.

The absence of critical edges in the complete model can in practice not be checked by a type checker because it requires knowing the complete model of binding upfront. We solve this by weakening the condition to a *sufficient* condition that can be checked. We then impose a well-formedness judgment on Statix-core specifications to also make this tractable in practice. Specifically, typing rules must have *permission to extend* a scope in the scope graph to be able to make assertions on the scope graph. In practice this means that although scopes can be queried from anywhere, they can only be extended locally with new binding information.

We prove that the operational semantics of Statix-core using the weaker sufficient condition is *sound* for well-formed specifications—i.e., it computes a type assignment and scope graph model that satisfy the specification. Importantly, and in contrast to the implementation of Statix by van Antwerpen et al. (2018), the non-deterministic operational semantics can also be proven *confluent* for the refined Statix-core language. The confluence argument again uses critical edges to reason about stability of query answers.

We implement the operational semantics and the static analysis that checks if all rules have sufficient permissions to extend scopes in Haskell. We give specifications of subsets of Java and Scala in Statix-core (extended with recursive predicates). Using these specifications we also *test* soundness of the reference implementation against the Java and Scala type checker. These case studies provide evidence of the expressiveness of Statix as a formalism, and show that the well-formedness restriction does not prohibit specifications of complex, real-world binding patterns.

In summary, the contributions of this paper are:

- A semantic characterization of name resolution query answer stability in terms of *critical edges* in an incomplete scope graph (§4.5.2).
- Statix-core (§4.3), a constraint language with built-in support for scope graphs, which distills and refines the core aspects of the Statix language and its declarative semantics due to van Antwerpen et al. (2018).
- An operational semantics for Statix-core (§4.4 and §4.5) that schedules name resolution queries such that query answer stability is guaranteed, thereby allowing language designers to abstract from the accidental complexity of implementing name resolution.
- A proof that the operational semantics of Statix-core is sound w.r.t. the declarative semantics of Statix-core (§4.5.3). The key that enables this proof is a type system for Statix-core (based on *permission to extend* a scope) and the scheduling criterion that is built into the operational semantics of Statix-core (based on an over-approximation of critical edges).
- MiniStatix, a Haskell implementation of Statix-core extended with (recursive) predicates. The implementation infers whether specifications have sufficient permissions to extend scopes, and can type check programs against their declarative language specification.
- Three case studies (§4.6) of languages specified in MiniStatix: (1) a subset of Java that includes packages, inner classes, type-dependent name resolution of fields and methods; (2) a subset of Scala with imports and objects; and (3) an implementation of the LMR module system that is similar to the one in Rust. The case studies demonstrate the expressive power and declarative nature of Statix-core, and test the approach against the reference type-checkers of Java and Scala.

4.2 Specifying & Scheduling Name Resolution

Programming languages with modules or objects (e.g., ML, Java, C♯, Scala, or Rust) use very different name resolution rules than languages with only lexical scoping. For example, the static semantics of non-lexical static binding, such as accessing a member of an object `o.m`, is to resolve the name `m` not in the *local* (lexical) scope, but in a *remote* scope (in this case the inner scope of the class declaration that corresponds to the type of the reference o). Similarly, a name in Scala or Rust is not always resolved in the lexical scope, but sometimes in an explicitly imported module or object scope, whose definitions may be declared in a very different part of the program.

These richer scoping constructs lead to more subtle resolution and disambiguation rules. Scala, for example, applies different scoping rules for names defined in the lexical scope (which can be forward referenced) compared to names that are imported (which cannot). Scala also applies different precedence rules depending on whether an imported name is explicitly listed, or caught by a wildcard. Precedence rules are often incomplete, in the sense that overlapping names sometimes lead to ambiguous uses. This requires more information to be available in environments.

These aspects make it more difficult to both *specify*, and *implement* static semantics. In this section we discuss both specification and implementation. We first discuss the role of name binding in the specification of static semantics (§4.2.1), and how Statix as a formalism enables the high-level specification of the above mentioned eatures (§4.2.2). We then discuss how name binding features contribute to a scheduling problem for type checkers (§4.2.3). Finally, we show how the innovative features of Statix impact this scheduling problem (§4.2.4). We will argue that there are two sides to this. On the one hand, these features make the scheduling problem more difficult because value dependencies are less explicit. On the other hand, the high level specification of binding in Statix provides a semantic tool to think about the scheduling problem and recover a provably sound schedule: critical edges. We end this section with an overview of how we use critical edges to address the scheduling problem for Statix.

4.2.1 Name Resolution: Non-lexical Static Binding and Disambiguation

The presence of non-lexical name binding can easily complicate a specification, harming conciseness, understanding, and maintenance of the static semantics rules. Typing rules use *type environments* to propagate binding information through a program. Type environments are appropriate and easy to use in the specification of static semantics for languages with only lexical binding because lexical binding follows the nesting structure of the AST. This is not the case for languages with non-lexical static scoping, where binding information may flow through references (e.g., module imports), or against the nesting structure of the AST (forward references; Hedin, 2000).

To demonstrate the issues that arise in language specification, we consider a simple Scala program. The program in Fig. 4.2a is a well-typed Scala program with two methods in an object o that mutually refer to one another. To specify the static semantics of such a list of mutually recursive definitions, we can follow the style of the ML specification (Milner et al., 1997), which uses rules of the form $C \vdash e \Rightarrow E$, with C the type environment of the phrase e, and E the context generated by the phrase e.

```
object o {
  def f:Int = g;
  def g:Int = f
}
```

(a) Mutual binding.

$$\textsc{T-Body}\;\frac{E + E' \vdash bs \Rightarrow E'}{E \vdash \{\, bs \,\} \Rightarrow E'} \qquad \textsc{T-Seq}\;\frac{E \vdash b \Rightarrow E' \qquad E \vdash bs \Rightarrow E''}{E \vdash b;bs \Rightarrow (E' \sqcup E'')}$$

$$\textsc{T-Def}\;\frac{E \vdash e : T}{E \vdash (\texttt{def}\, f \texttt{:}\ T \texttt{=} e) \Rightarrow \{f : T\}}$$

(b) Typing of mutual binding using environments.

Figure 4.2: Scala example program and the corresponding typing rules.

The context C is downward propagating, whereas E is upward propagating. We obtain the rules for block definitions shown in Fig. 4.2b. Name resolution behavior is the result of the way environments are combined in the different rules. The mutually-recursive behavior of the block is visible in rule T-Body, which *updates* the type environment with the aggregated binding that has propagated upwards from the block. The combination operator $+$ in the premise of T-Body updates the environment such that it shadows bindings in E that are also in E'. The disjoint union $\sqcup$ in the conclusion of T-Seq merges the environments produced by the definitions in the sequence, and enforces that the names do not overlap. We can see in this example that environments play two roles in these rules: to *aggregate* binding information from the program, and to *distribute* it throughout the program. Aggregation ties back into distribution at the scope boundary.

The update and disjoint union of environments are examples of bookkeeping operations that encode high-level binding concepts: disallowing duplicate definitions and shadowing respectively. Similarly, the 'cycle' in environment aggregation and distribution *encodes* mutual recursion. Encoding this using environments is a relatively small matter here, due to the limited number of rules and binding features to take into account. This becomes increasingly more difficult when we add language features that interact with binding and that require more sophisticated disambiguation.

In particular, non-lexical static binding complicates matters significantly: the definitions in Fig. 4.2a are not just locally in scope, but can be accessed from remote use sites, either qualified with the object `o`, or unqualified after importing object `o`. The potential for remote use significantly increases the required effort for aggregating and distributing binding facts. To lookup the structure of modules and classes, we may want to refer to a symbol table. Thus we have to explain through our typing rules how declarations generate unique entries in this symbol table. This requires aggregating all the entries to the root of the program. For the purposes of disambiguation, we may also need more structure in the environment. In Scala for example, we need to look beyond the closest matching binding because additional binders in outer scopes may make a reference ambiguous.

We argue that bookkeeping of environments is not a high-level means for expressing name resolution concepts of languages like Scala. Consequently, it is both unnecessarily hard to define rules that express the right semantics, and unnecessarily difficult to understand the high-level concepts from the written rules. Previous work proposes Statix (van Antwerpen et al., 2018) to address this problem. In §4.3, we discuss the concepts of Statix. We will show how Scala's name resolution rules can be understood using scope graphs, and made precise using Statix rules.

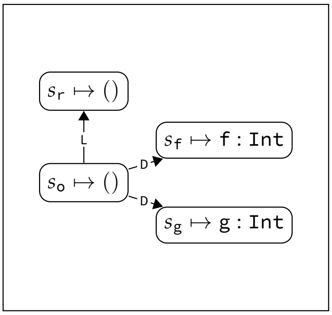
4.2.2 Declarative Specification using Scope Graphs in Statix

The problem of aggregating and distributing binding information is addressed by Statix in two ways: 1. scopes have independent existence and can be passed around, which allows extending scopes without the need for explicit aggregation, and allows remote access without explicit distribution; and 2. shadowing behavior is specified at the use site, allowing definitions to simply assert the scoping structure without having to anticipate all possible uses. To achieve this, Statix typing rules are predicates on terms and an *ambient scope graph*. Nodes in the graph represent scopes and binders, whereas (labeled) edges are used to represent (conditional) scope inclusion. Nodes contain a data term that can carry the information of a binder.

The binding of the program in Fig. 4.2a can be summarized as the scope graph in Fig. 4.3a. We write $s \mapsto t$ for a node with identity s and data term t. The nodes s_{R} and s_{O} represent the root scope and the object scope respectively. The latter is a lexical child of the former, indicated by the `L`-edge. The object scope contains two declarations, indicated by the two `D`-edges to declaration nodes, whose data terms `f:Int` and `g:Int` contain the usual information about the binders.

Previous work has shown how scope graphs can be used to model many binding structures (Néron et al., 2015; van Antwerpen et al., 2018; van Antwerpen et al., 2016). The fact that this particular scope graph models the binding of the given program, is made formal through a number of Statix rules, together with the declarative semantics of Statix. We give the required rules here using the Statix-core syntax, so that we can informally discuss how Statix constraints address the problems with declarative specification of binding using environments explained above. We will explain the formal syntax and declarative semantics of Statix-core in §4.3.

The Statix-core counterparts to the ML-specification style rules for the mutual binding in Fig. 4.2a are given in Fig. 4.3b. The Statix specification consists of constraint rules, which define that the typing judgment in the conclusion holds if the constraints in the premises hold. The phrases are typed in a lexical scope s, written



(a) Scope graph for Fig. 4.2a.

T-Body
$$\frac{(\nabla s' \mapsto ()) * (s' \xrightarrow{\mathsf{L}} s) * (s' \vdash bs)}{s \vdash \{\ bs\ \}}$$

T-Seq
$$\frac{(s \vdash b) * (s \vdash bs)}{s \vdash b;bs}$$

T-Def
$$\frac{\begin{array}{r}(s \vdash e : T) * (\nabla s' \mapsto (f : T)) \\ * \ (s \xrightarrow{D} s') * \mathsf{noDups}(s, f, s')\end{array}}{s \vdash (\texttt{def f : T = e})}$$

(b) Typing rules using Statix-core constraints.

Figure 4.3: Scope graph and Statix-core rules for the example in Fig. 4.2.

suggestively as $s \vdash t$.[3] Premises are separated using conjunction ($*$). The fact that blocks introduce new scope is expressed in the rule T-Body by asserting a scope s' in the scope graph (using $\nabla s' \mapsto ...$), connected to the lexical parent by an L-edge (using $s' \xrightarrow{\mathsf{L}} s$). The declarations are asserted similarly in the rule T-Def using a D-edge.

The first notable difference with the ML-style rules is that the Statix rules have no upward propagating context for aggregating binding. This is unnecessary because of the reference semantics of scopes in Statix rules. The rule T-Def can directly assert the structure that a definition induces in the ambient scope graph. Because the scope graph is a global model of binding, this structure does not need to be explicitly aggregated or distributed.

The second difference is in the way that lexical shadowing is specified. Rather than encoding this disambiguation rule using environment update in T-Body, the Statix-core rule only witnesses the structure of the scope graph model. Disambiguation is expressed directly in the rule for typing variables. We postpone the discussion of scope graph *queries* that fulfill this purpose until §4.3. For now it suffices to know that variable lookup works by finding minimal paths in the scope graph. Shadowing can be expressed by using a lexicographical path order where $\mathsf{D} < \mathsf{L}$.

The third difference is that the rule T-Seq is a completely binding-neutral rule. The fact that definitions should be unique in their scope, is expressed directly as a premise noDups(...) on the rule T-Def, rather than being encoded in the way that sequencing aggregates binders. We leave the predicate abstract for now, but it is specified using a graph query in the declaration scope.

Specification of languages with rich, non-lexical name binding features is complicated when using environment-based typing rules. **Statix provides a general formalism that allows concise specification of these languages, by removing the concerns of aggregating and distributing binding information from the typing rules.**

[3] The form of this typing judgment is not enforced in Statix rules—i.e., Statix predicates do not have to be defined exclusively over *AST* terms and can have multiple scope arguments.

4.2.3 Sound Type Checkers Require Scheduling

We now turn to the problem of writing a type checker based on a specification of static semantics, focusing on the difficulties surrounding name binding features. We will argue that type checkers face a *scheduling* problem in constructing the relevant environment and symbol table (or scope graph) to be able to type the names used in a program. Consider again the typing rules in Fig. 4.2b. A type checker arriving at the block faces the problem that the downward propagating input environment is constructed from the upward propagating output environment. For this reason, the type checker needs to be *staged*: it first needs to aggregate the binding from the block, before it can type check the expressions in the right environment. This simple example demonstrates how name binding induces dependencies between tasks in a type checker. Name resolution (and thus type checking) is only sound with respect to the typing rules if queries are only executed after all relevant information has been aggregated.

The binding features of a language determine how difficult it is to find a sound schedule. A language with forward references requires a schedule in which binding aggregation happens before querying. In our simple example, the schedule can be entirely static: one can always collect all definitions before ever typing their bodies. First class modules and type-dependent name resolution require more dynamic scheduling. For example, the resolution of a member name `m` in a Java or Scala expression `e.m(...)` requires the type of `e`. Typing `e` can in turn depend on all kinds of name resolution and type-checking tasks. This means that name resolution cannot be statically stratified.

When language engineers develop a type checker for a given language, they implement either such a statically stratified schedule as a number of fixed type-checking passes, or implement a method that in effect schedules type-checking tasks dynamically (even if the scheduling is simply 'on demand'). Soundness of the implemented approach is judged by the language engineers. Our goal is to *automatically* obtain sound type checkers from typing rules, and therefore we need a *systematic* approach to solving the scheduling problem.

4.2.4 Sound Schedules from Statix Rules

In §4.2.3 we arrived at a sound schedule for the typing of mutually recursive binding simply by lazily following the *demand* for dependencies. These dependencies are explicit in the environment-based rules of Fig. 4.2b. In languages with more complex scope and disambiguation rules, the dependencies of name resolution are not as easy to determine. We have argued that environment-based rules are difficult to specify for such languages. Ensuring that those rules can be evaluated on demand puts additional requirements on the rules, making it even more difficult to write the specification (J. T. Boyland, 2005). (This is a known problem with canonical attribute grammars. We compare in depth to attribute grammars in §4.7.) By decoupling scope from binding and name resolution rules in those scopes, Statix rules can specify complicated languages without regard for dependencies. As a result, more work is required to reconstruct the dependencies and a sound schedule from the rules.

We illustrate this with the Scala program in Fig. 4.4, which combines mutually recursive definitions with imports. The semantics of Scala are such that the definitions in an object are mutually recursive, allowing the forward reference g, while imports are sequential, only allowing references to the imported name h after the import statement. Local definitions have precedence over names imported in the same block, regardless of the order in which the definitions and imports appear in the program.

```
object o {
  def f:Int = g;
  import n._;
  def g:Int = h
}
object n {
  def h:Int = 42;
}
```

Figure 4.4: Scala example with mutual binding and imports.

The scoping structure of our example is modeled with the scope graph shown in Fig. 4.5. The dotted boxes show in which scopes names are resolved, with arrows indicating the resolution path. The definitions f and g are declared in the object scope s_o. Because imports are treated sequentially, import statements induce a scope, connected to the previous import or object scope using a B-edge. The import is represented by an I-edge to the scope s_n of object n. The forward reference g resolves to the definition in the same scope. The reference to h reaches the imported name via the B-edge and the outgoing I-edge.

Name resolution can be specified in terms of queries on the scope graph, which specify *reachability* and *visibility* of declarations in terms of a regular expression and an order on paths, respectively (§4.3.1). In this Scala subset, a declaration is reachable if it can be found in the scope graph via a path that matches the regular expression $\texttt{B}^*(\texttt{LB}^*)^*\texttt{I}^?\texttt{D}$. One can check that all the paths indeed match the regular expression.

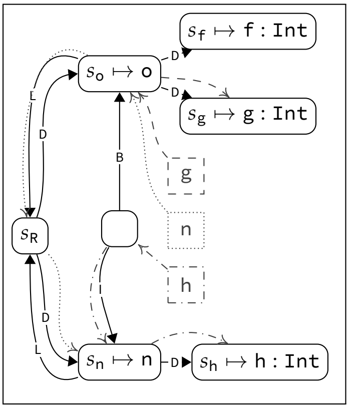


Figure 4.5: Scope graph corresponding to the program in Fig. 4.4.

During type checking, the scope graph is constructed from an initial empty graph, by adding more and more scopes and edges, until the graph is a complete model of the binding and scoping structure in the program. Name resolution is finding least reaching paths in the scope graph. Although conceptually simple, difficulty arises because scope graph construction can depend on resolving queries as well as the other way around. This is the case for imports, where the I-edge depends on resolution of the named import. In general, even the fact whether there is an edge *at all* can depend on name resolution. This means that scope graph construction *must* be interleaved with query evaluation.

This raises the following concrete scheduling problem: Given a scope graph query, a partial scope graph, and a partially satisfied type specification, is it sound to evaluate the query now or should it be delayed? Conceptually, the answer is 'yes, it is sound' if the answer to the query in the current partial model is the same as the answer in a complete model. The answer is 'no, delay' if the complete model contains additional binding information that is relevant to the query at hand.

To specify what information is relevant, we introduce the notion of *critical edges* for a query in a model with respect to a partial scope graph. An unstable resolution answer means that a resolution path that is valid in the model graph is not yet a valid path in the partial graph because some part of the final graph is missing. A *critical edge* of a query is an edge along a resolution path in the model that is not present yet in the partial graph, but whose source node is present. We can think of critical edges as the root cause of instability, as they are the *first* missing step in a resolution path in the model. Whether an edge is critical is determined based on the regular expression that expresses reachability, which exactly demarcates the part of the scope graph that will be searched.

Because the complete model is yet unknown, we cannot directly identify missing critical edges. Instead, we look ahead at the remaining type checking problem to determine whether any critical edges are still missing. In general, precise determination may require arbitrary type checking, which would lead to a backtracking implementation. Instead, we approximate critical edges as *weakly* critical edges, whose absence can be determined without backtracking. We show that our approximation is sound for a subset of Statix specifications. Importantly, we can statically determine if a specification is in this subset using a type analysis that we formalize as *permission-to-extend*.

4.3 Statix-core: A Constraint Language

In this section we introduce Statix-core, modeling the essential ingredients of Statix (van Antwerpen et al., 2018): a framework for the declarative specification of type systems. Statix specifications have a precise declarative semantics that specifies which scope graphs are models of the specification. They do not have a formal operational semantics that can be used to find a model for a given program if it exists. Such an operational semantics requires a sound scheduling strategy for name and type resolution.

In §4.3.1 we first introduce scope graphs formally, together with a concise presentation of its resolution calculus (Néron et al., 2015; van Antwerpen et al., 2016). We then present the syntax (§4.3.2) and declarative semantics (§4.3.3) of Statix-core. Subsequently, in §4.4 and §4.5, we present the sound operational semantics using a general delay mechanism for queries based on critical edges.

4.3.1 Preliminaries

Statix-core is a constraint language extended with primitives for *scope graph assertions and queries*. The assertions internalize scope graph construction, whereas the queries internalize scope graph resolution. We discuss what a scope graph comprises, and present resolution in scope graphs as computing the answer to a *visibility query*.

Scope graphs A scope graph $\mathcal{G}$ is a triple $\langle S, E, \rho \rangle$ where S is a set of *node identifiers*, E is a multi-set of *labeled, directed edges*, and ρ is a *finite map* from node identifiers to terms. We will write $S_\mathcal{G}$, $E_\mathcal{G}$ and $\rho_\mathcal{G}$ for projecting the three components out of a graph $\mathcal{G}$, and may omit the subscript when it is unambiguous. We will refer to the term associated with a node identifier as the *datum* of a node. The complete syntax of graphs and terms is given in Fig. 4.6. We write ϵ for the empty graph and $\mathcal{G} \sqsubseteq \mathcal{G}'$ for the extension order on graphs. On sets we use the notation $X \sqcup Y$ to denote the *disjoint union* of sets X and Y, $X \setminus Y$ to denote the set difference, and $x; X$ to denote $\{x\} \sqcup X$.

Regular paths. Name resolution is modeled with *regular paths* in the graph. We write $\mathcal{G} \vdash p : s \xrightarrow{w} s_k$ to denote that p is a regular (acyclic) path in $\mathcal{G}$, starting in s, ending in s_k, and spelling the word w along its edges. We define the operations $\mathsf{src}\ (\_)$, $\mathsf{tgt}\ (\_)$ and $\mathsf{labels}\ (\_)$ to act on paths and project out the source node s, target node s_k, and list of labels on the edges, respectively.

Graph queries. The answer to a *reachability query* $s \overset{r}{\twoheadrightarrow} D$ is a set of regular paths $s \xrightarrow{w} s'$ such that w matches the regular expression r and the datum of s' inhabits the term predicate D:

$$\mathsf{Ans}\left(\mathcal{G}, s \overset{r}{\twoheadrightarrow} D\right) = \left\{p \;\middle|\; \mathcal{G} \vdash p : s \xrightarrow{w} s' \text{ and } w \in \mathcal{L}(r) \text{ and } \rho_\mathcal{G}(s') \in D\right\}$$

We write $\mathcal{L}(r)$ for the set of words in the regular language described by r. A useful device when we consider partial reaching paths is the Brzozowski derivative (Brzozowski, 1964) $\delta_w r$ of a regular expression r with respect to a word w, whose language is $\mathcal{L}(\delta_w r) = \{w' \mid ww' \in \mathcal{L}(r)\}$.

Often we are interested in a refinement of reachability, which we call *visibility*. A datum is visible via a path p only if p is a least reaching path. Given reachability answer A, the subset of visible paths is defined as the minimum of A over a preorder R on paths:

$$\mathsf{min}(A, R) = \{p \in A \mid \forall q \in A.\, Rqp \Rightarrow Rpq\}$$

Reachability is monotone with respect to graph extension: extending a graph with additional nodes and edges can only make *more* things reachable. In contrast, visibility is *non-monotonic* with respect to graph extension: extending a graph with additional nodes and edges may obscure—i.e., shadow—information that was previously visible.

We can now formally state the notion of stability of query answers that is key to the correct implementation of static name resolution: a query (answer) q is said to be *stable* between graphs $\mathcal{G} \sqsubseteq \mathcal{G}'$, when the answer set for the query is identical in both graphs: i.e. $\mathsf{Ans}\,(\mathcal{G}, q) = \mathsf{Ans}\,(\mathcal{G}', q)$.

4.3.2 Syntax of Statix-core

We introduce the constraint language Statix-core for making assertions about terms and an implicit, ambient scope graph. The syntax is defined in Fig. 4.6. We summarize the main syntactic categories.

Terms t are either variables x, compound terms $f(t^*)$, graph edge-labels l, graph nodes s, or graph edges $t \xrightarrow{l} t$. Importantly, nodes only appear as an artifact of substitution in the operational semantics and do not appear in source constraint problems. Literals for sets of terms $\bar{t}$ are used to represent query answer sets in programs and are generated from the disjoint union of singletons and empty sets. Sets of terms are implicitly understood to exist up to reordering.

Constraints C define assertions on terms and an underlying scope graph. As we shall see in §4.3.3, constraint satisfaction uses a notion of *ownership*, which gives the semantics a separation logic (O'Hearn et al., 2001) flavor. This is reflected in the syntax of Statix-core where we use $C * C$ for *separating* conjunction, and emp and false for the neutral and absorbing elements of $*$, respectively. The $t_1 = t_2$ constraint asserts that t_1 and t_2 are equal. The x binder in existential quantification $\exists x.C$ ranges over all possible terms, whereas the x in universal quantification $\forall x \;\mathsf{in}\; \bar{t}.C$ ranges over members in a given finite set of terms $\bar{t}$.

The assertions on the ambient scope graph $\mathcal{G}$ come in two flavors: node and edge assertion. The former is written $\nabla t_1 \mapsto t_2$ and assert that t_1 is a node $s \in S_{\mathcal{G}}$ such that $\rho_{\mathcal{G}}(s) = t_2$. The node assertion gets unique ownership of s, such that no other node assertion can observe the same fact about the model $\mathcal{G}$. Similarly, edge assertions $t_1 \xrightarrow{l} t_2$ assert unique ownership of an edge $(t_1, l, t_2) \in E_{\mathcal{G}}$. The $\mathsf{dataOf}(t_1, t_2)$ constraint asserts that the data associated with node t_1 is t_2.

Query constraints ($\mathsf{query}\; t \xrightarrow{r}\!\!\!\!\rightarrow D \;\mathsf{as}\; z.C$) internalize the reachability queries from §4.3.1: we query node t for the set of all reaching paths over the regular expression r

Signature

l	$\in$	$\mathcal{I}$	label
f	$\in$	$\mathcal{F}$	term constructor symbol
r	$\in$	$\mathcal{R}$	regular expression

Terms

$t \in \mathcal{T}$	$::=$	x	variable
	$\mid$	$f(t^*)$	compound term
	$\mid$	$l \mid s$	label and node

Variables

x	$\in$	$\mathcal{X}$	term variable
z	$\in$	$\mathcal{Z}$	set variable
s	$\in$	$\mathcal{V}$	node name

Sets of Terms

$\bar{t}$	$::=$	z	set variable
	$\mid$	ζ	set literal
ζ	$::=$	$\emptyset$	empty set
	$\mid$	$\{t\}$	singleton set
	$\mid$	$\zeta \sqcup \zeta$	disjoint union

Graphs

$$\mathcal{G} \quad ::= \quad \langle S \subseteq \mathcal{V}, \quad E \subseteq (\mathcal{V} \times \mathcal{I} \times \mathcal{V}), \quad \rho \subseteq (\mathcal{V} \rightharpoonup \mathcal{T})\rangle$$

Constraints

C	$::=$	$\mathsf{emp} \mid \mathsf{false}$	true and false
	$\mid$	$C * C$	separating conjunction
	$\mid$	$t = t \mid \exists x.C$	term equality and quantification
	$\mid$	$\mathsf{single}(t, \bar{t}) \mid \forall x \text{ in } \bar{t}.C$	set singletons and quantification
	$\mid$	$\mathsf{min}(\bar{t}, R, \bar{t})$	set minimum
	$\mid$	$\nabla t \mapsto t \mid t \xrightarrow{l} t$	node and edge assertion
	$\mid$	$\mathsf{query}\ t \xrightarrow{r}\!\!\!\!\rightarrow D \text{ as } z.C \mid \mathsf{dataOf}(t, t)$	graph query and data retrieval

Figure 4.6: Syntax of Statix-core.

to nodes whose data satisfy the predicate D, and bind the query result to z in C. Queries yield *sets* of paths (embedded as terms) which motivates the need for set literals, forall quantification over these, and the $\mathsf{single}(t, \bar{t})$ constraint which asserts that $\bar{t}$ is a singleton set containing just the element t. The constraint $\mathsf{min}(\bar{t}, R, \bar{t}')$ asserts that the latter set of terms is the minimum of the former over the preorder R and is used to specify disambiguation of a set of reaching paths to the set of visible paths. We implicitly convert between mathematical sets and term set syntax where necessary. We assume that the set $\mathcal{F}$ of term constructor symbols contains the necessary constructors to encode paths.

4.3.3 Declarative Semantics of Statix-core

The meaning of constraints is given by the *constraint satisfaction* relation that is inductively defined by the rules in Fig. 4.7. Satisfiability is expressed as $\mathcal{G} \vDash_\sigma C$, stating that the graph $\mathcal{G}$ satisfies the closed constraint C with *graph support* $\sigma = \langle S, E\rangle$, where $S \subseteq S_\mathcal{G}$ and $E \subseteq E_\mathcal{G}$. In case the satisfaction judgment holds, we say that $\mathcal{G}$ is a *model* for the constraint C.

We lift the declarative semantics to open constraints in the usual way and write $\mathcal{G}, \varphi \vDash_\sigma C$ to denote $\mathcal{G} \vDash_\sigma C\varphi$. We also define constraint entailment $\Vdash$ and equivalence $\dashv\Vdash$, which we will use when we consider the properties of the operational

$\mathcal{G} \vDash_\sigma C$ Scope graph $\mathcal{G}$ satisfies constraint C with support σ

$$\textsc{Emp}\ \frac{}{\mathcal{G} \vDash_\perp \mathsf{emp}} \qquad \textsc{Conj}\ \frac{\mathcal{G} \vDash_{\sigma_1} C_1 \qquad \mathcal{G} \vDash_{\sigma_2} C_1}{\mathcal{G} \vDash_{\sigma_1 \sqcup \sigma_2} C_1 * C_2} \qquad \textsc{Eq}\ \frac{t_1 = t_2}{\mathcal{G} \vDash_\perp t_1 = t_2} \qquad \textsc{Exists}\ \frac{\mathcal{G} \vDash_\sigma C\,[t/x]}{\mathcal{G} \vDash_\sigma \exists x.C}$$

$$\textsc{Singleton}\ \frac{}{\mathcal{G} \vDash_\perp \mathsf{single}(t, \{t\})} \qquad \textsc{Min}\ \frac{\bar{t}' = \mathsf{min}(\bar{t}, R)}{\mathcal{G} \vDash_\perp \mathsf{min}(\bar{t}, R, \bar{t}')} \qquad \textsc{Forall-Empty}\ \frac{}{\mathcal{G} \vDash_\perp \forall x \text{ in } \emptyset.C}$$

$$\textsc{Forall}\ \frac{\mathcal{G} \vDash_{\sigma_1} C\,[t_1/x] \qquad \mathcal{G} \vDash_{\sigma_2} \forall x \text{ in } \bar{t}_2.C}{\mathcal{G} \vDash_{\sigma_1 \sqcup \sigma_2} \forall x \text{ in } (\{t_1\} \sqcup \bar{t}_2).C} \qquad \textsc{Node}\ \frac{s \in S_{\mathcal{G}} \qquad \rho_{\mathcal{G}}(s) = t}{\mathcal{G} \vDash_{\langle s, \emptyset \rangle} \nabla s \mapsto t} \qquad \textsc{Edge}\ \frac{(s_1, l, s_2) \in E_{\mathcal{G}}}{\mathcal{G} \vDash_{\langle \emptyset, (s_1, l, s_2) \rangle} s_1 \xrightarrow{l} s_2}$$

$$\textsc{Query}\ \frac{\mathcal{G} \vDash_\sigma C\left[\mathsf{Ans}\left(\mathcal{G}, s \overset{r}{\twoheadrightarrow} D\right)/z\right]}{\mathcal{G} \vDash_\sigma \mathsf{query}\ s \overset{r}{\twoheadrightarrow} D \text{ as } z.C} \qquad \textsc{Data}\ \frac{\rho_{\mathcal{G}}(s) = t}{\mathcal{G} \vDash_\perp \mathsf{dataOf}(s, t)}$$

Figure 4.7: Statix constraint satisfiability.

semantics:

$$\textsc{Entails}\ \frac{\forall \mathcal{G}, \varphi, \sigma.\, (\mathcal{G}, \varphi \vDash_\sigma C_1 \text{ implies } \mathcal{G}, \varphi \vDash_\sigma C_2)}{C_1 \Vdash C_2} \qquad \textsc{Equivalent}\ \frac{C_1 \Vdash C_2 \qquad C_2 \Vdash C_1}{C_1 \dashv\Vdash C_2}$$

The graph support declaratively expresses *ownership* of graph structure in constraints. The role of support in constraint satisfiability gives the resulting logic a separation logic flavor. Support is distributed linearly, which means that we get the constraint equivalences of linear logics: conjunction is commutative and associative and has emp as its identity and false as the absorbing element, but the left and right elimination rules of conjunction do not hold.

We lift set operations pointwise to graph support. A particularly important operation is the disjoint union, written $\sigma_1 \sqcup \sigma_2$, which is defined as $\sigma_1 \cup \sigma_2$, if and only if $\sigma_1 \cap \sigma_2$ is empty. We write $\perp$ to denote empty support and distinguish fully supported models from unsupported ones:

$$\frac{\mathcal{G} \vDash_{\langle S_{\mathcal{G}}, E_{\mathcal{G}} \rangle} C}{\mathcal{G} \vDash C}\ \textsc{Supported}$$

Intuitively, a model $\mathcal{G}$ is supported by a constraint C when every node and edge in it is asserted by C. For top-level constraints, we are exclusively interested in supported models. Models that are *not fully supported* at the top-level contain "junk": graph structure that is not asserted by the Statix specification. For our problem domain it does not make sense to consider those models, as they would contain binding structure that does not correspond to the input program. Not every constraint that has a model also has a supported one. Consider for example the following constraint:

$$\exists s.\ \Big(\mathsf{query}\ s \xrightarrow{P*}\!\!\!\!\rightarrow D\ \mathsf{as}\ z.\ (\exists x.\mathsf{single}(x, z))\Big)$$

Whenever D is inhabited, there are clearly graphs that satisfy the constraint. None of those graphs are supported, however, because there are no node or edge assertions. This means that the whole constraint has empty support and the empty graph is not a model of the query.

4.4 Solving Constraints

Our goal is to derive, from the Statix specification of a type system, an executable type checker. A *sound* type checker should take a specification and an input program e and *construct* the ambient scope graph $\mathcal{G}$ such that $\mathcal{G}$ and e together obey the specification. Or, if and only if the program does not obey the specification, produce an error. Our approach to this is to equip Statix-core with an operational semantics that reduces constraints, as generated over a program, to a graph that satisfies the constraint according to the declarative semantics, or rejects the constraint *if and only if* such a graph does not exist. In this section we describe such an operational semantics *without queries*. We show that the operational semantics enjoys confluence and soundness with respect to the declarative semantics. Extending the operational semantics to queries requires us to schedule constraint solving such that the (implicit) dependencies between graph construction and query resolution are appropriately respected. In §4.5 we formally discuss a naive, unsound strategy, and develop a sound strategy derived from a formal characterization of a criterion for answer stability: absence of critical edges in graph extensions.

4.4.1 The Small-Step Operational Semantics

The operational semantics of Statix without queries is a small-step semantics defined on state tuples $\langle \mathcal{G} \mid \overline{C} \rangle$, where $\mathcal{G}$ is a graph and $\overline{C}$ is a set of constraints that is repeatedly simplified. The interesting rules are displayed in Fig. 4.8. The full operational semantics can be found in Rouvoet, van Antwerpen, et al. (2020c, Appendix B). Semantically we treat the constraint set as a large conjunction and we non-deterministically pick a constraint from this set to perform a step on.

A constraint C is solved by constructing an initial state κ as $\langle \epsilon \mid \{C\}\ \rangle$ and repeatedly stepping until a final or stuck state κ' is reached. We say that the operational semantics *accepts* C iff it reaches a final state $\langle \mathcal{G} \mid \emptyset \rangle$ and *rejects* C iff it reaches a final state $\langle \mathcal{G} \mid \{\mathsf{false}\}\ \rangle$. Any other states in which we cannot reduce by taking a step are said to be *stuck*.

$\kappa \to \kappa'$ State κ steps to κ'

Op-Conj

$$\langle \mathcal{G} \mid (C_1 * C_2)\,;\overline{C}\rangle \to \langle \mathcal{G} \mid C_1;C_2;\overline{C}\rangle$$

Op-Eq-True

$$\frac{t_1\varphi = t_2\varphi \qquad \varphi \text{ is most general}}{\langle \mathcal{G} \mid (t_1 = t_2)\,;\overline{C}\rangle \to \langle \mathcal{G}\varphi \mid \overline{C}\varphi\rangle}$$

Op-Eq-False

$$\frac{\neg\exists\varphi.t_1\varphi = t_2\varphi}{\langle \mathcal{G} \mid (t_1 = t_2)\,;\overline{C}\rangle \to \langle \mathcal{G} \mid \{\mathsf{false}\}\ \rangle}$$

Op-Exists

$$\frac{y \text{ is fresh for } \mathcal{G} \text{ and } \overline{C}}{\langle \mathcal{G} \mid (\exists x.C)\,;\overline{C}\rangle \to \langle \mathcal{G} \mid C\,[y/x]\,;\overline{C}\rangle}$$

Op-Singleton-True

$$\langle \mathcal{G} \mid \mathsf{single}(t, \{t'\});\overline{C}\rangle \to \langle \mathcal{G} \mid (t = t');\overline{C}\rangle$$

Op-Singleton-False

$$\frac{\neg\exists t'.\bar{t} = \{t'\}}{\langle \mathcal{G} \mid \mathsf{single}(t, \bar{t});\overline{C}\rangle \to \langle \mathcal{G} \mid \{\mathsf{false}\}\ \rangle}$$

Op-Node-Fresh

$$\frac{s \notin S}{\langle\langle S, E, \rho\rangle \mid (\nabla x \mapsto t);\overline{C}\rangle \to \langle\langle (s;S), E, \rho[s \to t]\,[s/x]\rangle \mid \overline{C}\,[s/x]\ \rangle}$$

Op-Node-Stale

$$\frac{t_2 \text{ is not a variable}}{\langle \mathcal{G} \mid (\nabla t_2 \mapsto t_1)\,;\overline{C}\rangle \to \langle \mathcal{G} \mid \{\mathsf{false}\}\ \rangle}$$

Op-Data

$$\frac{\rho(s) = t_2}{\langle \mathcal{G} \mid \mathsf{dataOf}(s, t_1);\overline{C}\rangle \to \langle \mathcal{G} \mid (t_1 = t_2);\overline{C}\rangle}$$

Op-Edge

$$\langle\ \langle S, E, \rho\rangle \mid (s_1 \xrightarrow{l} s_2);\overline{C}\rangle \to \langle\langle S, (s_1, l, s_2)\,; E, \rho\rangle \mid \overline{C}\rangle$$

Figure 4.8: Operational semantics of Statix without queries (complete rules in Rouvoet, van Antwerpen, et al. (2020c, Appendix B)).

The rules for the usual logical connectives (emp, false, $C_1 * C_2$, $=$, $\exists$, $\forall$, and single) are standard. The rule for answer set minimums simply proceeds by computation. For $\nabla t_1 \mapsto t_2$ there are two rules. If t_1 is a variable x, rule Op-Node-Fresh will extend the graph with a fresh node s, claim unique ownership over it, and substitute s for x everywhere. If t_1 is not a variable, specifically if it is a node, then it must be owned already and the rule Op-Node-Stale rejects the constraint by stepping to $\{\mathsf{false}\}$. For example, both rules would be executed once for the specification $\nabla x \mapsto () * \nabla x \mapsto ()$: one of the constraints gets ownership, and the other fails to get it. Edge assertions $t_1 \xrightarrow{l} t_2$ construct new edges in the graph via Op-Edge when both endpoints have become nodes. Multiple edges with the same label between the same endpoints can exist separately—i.e., there is no edge counterpart to the be Op-Node-Stale rule. Data assertions $\mathsf{dataOf}(t_1, t_2)$ compute by unification when the node t_1 becomes ground.

4.4.2 Properties of the Operational Semantics

We will show that the operational interpretation of a Statix-core specification is sound with respect to the declarative reading. That is, if the operational semantics accepts a constraint C, then the resulting graph is a *supported model* for C. And additionally, if the operational semantics rejects a constraint C, then there exists no supported model for C. From the perspective of the object language semantics defined in Statix-core this means that the derived type-checker is sound by construction with respect to the typing rules of the language.

If we extend our declarative semantics for constraints to states, we can state the soundness criterion more concisely and uniformly. We accomplish this via an *embedding* of states into constraints:

Definition 1. The *embedding of a graph* $\langle V, E, \rho \rangle$ and the *embedding of a state* $\langle \mathcal{G} \mid \overline{C} \rangle$ are defined as follows:

$$[\![\langle V, E, \rho \rangle]\!] = \left(\mathop{\ast}_{s \in V} \nabla s \mapsto \rho(s) \right) * \left(\mathop{\ast}_{(s,l,s') \in E} (s \xrightarrow{l} s') \right)$$

$$[\![\langle \mathcal{G} \mid \overline{C} \rangle]\!] = [\![\mathcal{G}]\!] * (\ast \overline{C})$$

The soundness criterion can now be stated in terms of constraint equivalence between initial and final states. Specifically, we will show that the following theorem holds:

Theorem 1 (Soundness of Statix-core without queries)**.** *Let κ be either an accepting or rejecting state. The operational semantics for Statix-core without queries is* sound*:*

$$\langle \epsilon \mid \{C\} \rangle \to^* \kappa \textit{ implies } C \dashv\vdash [\![\kappa]\!]$$

This is equivalent to the aforementioned informal definition of soundness, which can be shown using the facts that top-level constraints are closed and that graphs are trivially a model for their own embedding. We would like to prove this statement by induction on the trace of steps. This requires us to show that individual steps operate along constraint equivalences—i.e., that $\kappa_1 \rightarrow \kappa_2$ implies $\llbracket \kappa_1 \rrbracket \mathrel{\dashv\Vdash} \llbracket \kappa_2 \rrbracket$. Indeed, this is the case for many of the rules. For example, Op-Conj and Op-Emp rewrite along commutativity, associativity, and identity of the separating conjunction. The rules for existential quantification and node assertion, however, cannot be justified using logical equivalences. To this end we define a more general notion of *preserving satisfiability*:

Definition 2. We write $C_1 \mathrel{|\!\sim} C_2$ to denote that C_2 is satisfiable when C_1 is satisfiable, that is, the existence of a model $\mathcal{G}$ for open constraint C_1, implies that $\mathcal{G}$ is also a model for C_2, but modulo graph equivalence ($\approx$):

$$\frac{\forall \mathcal{G}_1, \varphi_1.\ (\mathcal{G}_1, \varphi_1 \vDash C_1 \text{ implies } (\exists \mathcal{G}_2, \varphi_2.\, \mathcal{G}_2, \varphi_2 \vDash C_2 \quad \text{s.t. } \mathcal{G}_1 \approx \mathcal{G}_2))}{C_1 \mathrel{|\!\sim} C_2}$$

We also define the symmetric counterpart $C_1 \mathrel{\sim\!|\!|\!\sim} C_2 \equiv C_1 \mathrel{|\!\sim} C_2 \land C_1 \mathrel{\sim\!|} C_2$, which denotes preservation of satisfiability. For top-level (closed) constraints this notion of preserving satisfiability coincides with constraint equivalence. Furthermore, constraint entailment $C_1 \Vdash C_2$ always implies $C_1 \mathrel{|\!\sim} C_2$, allowing the use of laws such as identity, commutativity, and associativity of the separating conjunction when we reason about preservation of satisfiability. Steps in the operational semantics are semantically justified in that they preserve satisfiability of the constraint problem:

Lemma 1. *Steps preserves satisfiability: $\kappa_1 \rightarrow \kappa_2$ implies $\llbracket \kappa_1 \rrbracket \mathrel{\sim\!|\!|\!\sim} \llbracket \kappa_2 \rrbracket$*

This may feel counter-intuitive, as steps construct a graph and preservation of satisfiability demands equivalent graphs as the model for the left- and right-hand-sides of the step. The key to understanding this lies in Def. 1 of the state embedding together with the rules for graph construction Op-Node-True and Op-Edge, which show that bits of graph (support) are *merely moved* between the constraint program and the (partial) model. In the initial state the entire model should be specified in the input constraint and in the final state the entire model is a given.

Proof sketch. The proof is by case analysis on the constraint that is the focus of the step. Many cases can indeed be proven using logical equivalences. Other cases, such as the elimination of existential quantifiers rely on the commutativity of substitutions with embedding of states. The graph equivalence is trivial everywhere, except for the step Op-Node-True. An arbitrary *fresh* node is chosen there, which means that the models for the different sides of the step are only equal up-to renaming of nodes. □

As a consequence of Lemma 1, the operational semantics enjoys soundness with respect to the declarative semantics (Thm. 1).

Proof sketch of Thm. 1. The embeddings of the initial and final states reduce to C and $[\![\mathcal{G}]\!]$ respectively. We repeatedly apply the fact that steps preserve satisfiability and prove $C \mathrel{\sim\!\mid\!\mid\!\!\sim} [\![\mathcal{G}]\!]$. Now we make use of the fact that graphs are trivially a supported model for their own embedding: $\mathcal{G} \vDash [\![\mathcal{G}]\!]$. By the above constraint equivalence, $\mathcal{G}$ must then also be a supported model for C up-to renaming of nodes. The theorem follows from the fact that constraint satisfaction is preserved by consistent renaming of nodes in the model and the constraint, and the fact that node renaming vanishes on top-level constraints. □

The operational semantics is non-deterministic, but confluent. This can be shown to hold by proving the diamond property for the reflexive closure of the step relation. A sketch of the proof can be found in the Rouvoet, van Antwerpen, et al., 2020c, Appendix A.

Theorem 2 (Confluence). *If $\kappa \rightarrow^* \kappa_1$ and $\kappa \rightarrow^* \kappa_2$ then there exists κ_1' and κ_2' such that $\kappa_1 \rightarrow^* \kappa_1'$ and $\kappa_2 \rightarrow^* \kappa_2'$ where $\kappa_1' \approx \kappa_2'$.*

4.5 Solving Queries: Knowing When to Ask

We address the problem of extending the Statix-core operational semantics to support queries. First we improve our understanding of the problem, by considering a naive semantics that answers queries unconditionally. We show that this approach yields unsound name resolution by violating answer stability. A rule for queries needs to ensure that query answers are stable. We develop the sound rule in three steps: (1) We characterize the scope graph extensions that causes query answer instability (§4.5.1) and show that we can guarantee stability by ensuring the *absence of (weakly) critical edge* extensions. (2) We describe a fragment of *well-formed* constraint programs for which it is feasible to check, without constraint solving, that certain graph edges cannot exist in any future graph (§4.5.3), addressing the problem that the complete scope graph is unknown during type checking. (3) We obtain an operational semantics for well-formed Statix-core constraints *with queries* by *guarding* query simplification by the absence of weakly critical edges in all future graphs. We prove that this guarded rule preserves satisfiability and thus yields a sound operational semantics (§4.5.3, Thm. 5). In §4.6 we discuss case studies we conducted to test the completeness of the operational semantics.

4.5.1 Naive Query Answering

Consider a naive and unconditional rule for queries $\mathsf{query}\ s \overset{r}{\twoheadrightarrow} D\ \mathsf{as}\ z.C$:

$$\frac{\bar{t} = \mathsf{Ans}\left(\mathcal{G}, s \overset{r}{\twoheadrightarrow} D\right)}{\left\langle \mathcal{G} \mid \mathsf{query}\ s \overset{r}{\twoheadrightarrow} D\ \mathsf{as}\ z.C; \overline{C} \right\rangle \to \left\langle \mathcal{G} \mid C\left[\bar{t}/z\right]; \overline{C} \right\rangle}\ \textsc{Op-Query-Naive}$$

It solves them by answering the given reachability query in the incomplete graph that is part of the solver state at that time. It then simplifies the constraint program by substituting the answer set into C. This rule is *unsound*: it results in graphs that are not models of the input constraint. Consider the following example:

$$\overline{C} = \nabla x \mapsto (); \nabla y \mapsto (); \mathsf{query}\ x \overset{P+}{\twoheadrightarrow} \top\ \mathsf{as}\ z.\left(\forall x'\ \mathsf{in}\ z.\mathsf{false}\right); x \xrightarrow{P} y$$

The query in these constraints asks for any node that is reachable in the graph after traversing at least one P-labeled edge, starting in the node for the variable x. It then asserts (via $\forall x'\ \mathsf{in}\ z.\mathsf{false}$) that the answer to this query is empty. A complete trace for this example is visualized in Fig. 4.9. Clearly, the final graph in Fig. 4.9 is *not* a model for the input constraint. The answer to the query in the final graph is non-empty: there is a single path in the answer consisting of the only edge in the graph. The reason for this faulty behavior can be reduced to two observations: (1) the naive solver answers queries based on incomplete information, namely the partial graph that happens to be part of its state at that point in the trace, and (2) query answers are in general not stable under graph extensions that occur later in the constraint solver. This raises the question: what additional conditions must hold in a given state such that query solving *is sound*—i.e., under what side-condition is the following rule for query answering sound?

$$\left\langle \mathcal{G} \mid \mathsf{query}\ s \overset{r}{\twoheadrightarrow} D\ \mathsf{as}\ z.C; \overline{C} \right\rangle \to \left\langle \mathcal{G} \mid C\left[\mathsf{Ans}\left(\mathcal{G}, s \overset{r}{\twoheadrightarrow} D\right)/z\right]; \overline{C} \right\rangle$$

In order to prove that this rule is sound, it suffices to prove that it preserves satisfiability, as is the case for the other steps of the operational semantics (c.f. Lemma 1). Concretely, to show that this rule preserves satisfiability, we have to prove:

$$[\![\mathcal{G}]\!] * \mathsf{query}\ s \overset{r}{\twoheadrightarrow} D\ \mathsf{as}\ z.C; \overline{C} * \left(\circledast \overline{C}\right) \dashv\Vdash\!\!\sim [\![\mathcal{G}]\!] * C\left[\mathsf{Ans}\left(\mathcal{G}, s_1 \overset{r}{\twoheadrightarrow} D\right)/z\right] * \left(\circledast \overline{C}\right)$$

This means that every supported model $\mathcal{G}'$ for the left constraint must be a supported model for the right constraint as well, and vice versa. When is this the case? It holds exactly when the query $s \overset{r}{\twoheadrightarrow} D$ is stable for the graph extension $\mathcal{G} \sqsubseteq \mathcal{G}'$. Or, in the terms of the application domain of Statix, it holds if all *relevant* namebinding information that may influence resolution of the specified name is present in $\mathcal{G}$. That means, for example, that no further names will be discovered in the remainder of the program that shadow declarations that are reachable in the current graph $\mathcal{G}$.

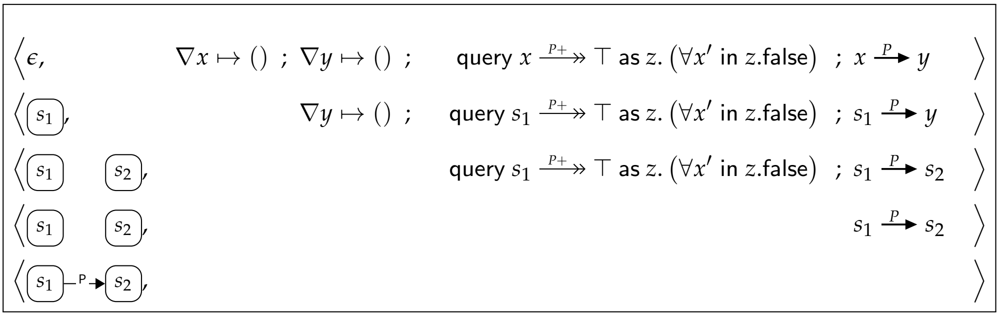

⟨ε, ∇x ↦ () ; ∇y ↦ () ; query x ↠ ⊤ as z. (∀x′ in z.false) ; x → y⟩
P+
P
⟨s1, ∇y ↦ () ; query s1 ↠ ⊤ as z. (∀x′ in z.false) ; s1 → y⟩
⟨s1 s2, query s1 ↠ ⊤ as z. (∀x′ in z.false) ; s1 → s2⟩
⟨s1 s2, s1 → s2⟩
⟨s1 –P→ s2,⟩

Figure 4.9: Trace demonstrating unsoundness of a naive query simplification rule.

4.5.2 Ensuring Answer Stability

In this section we untangle the definition of stability under graph extension and find the *root cause* of instability: *critical edges* in a scope graph extension. To guarantee query stability, we want to prevent that the solver extends the graph with critical edges. We argue however that the absence of critical edges is too strong a notion for a solver to verify. To remedy this, we derive the notion of a *weakly critical edge* which only considers the extension boundary.

To appoint a *root cause of instability* of reachability queries under graph extensions $\mathcal{G} \sqsubseteq \mathcal{G}'$, we focus on paths that exist in $\mathcal{G}'$, but not in $\mathcal{G}$:

$$p \in \mathsf{Ans}\left(\mathcal{G}', s_1 \xrightarrow{r}\!\!\!\!\!\rightarrow D\right) \setminus \mathsf{Ans}\left(\mathcal{G}, s_1 \xrightarrow{r}\!\!\!\!\!\rightarrow D\right)$$

Because the start node of every path in the answer set of a query is fixed (in this case to s_1) they can always be partitioned into a non-empty prefix in $\mathcal{G}$ and the remainder. The first edge of the remainder can be considered the root cause for this new path in $\mathcal{G}'$. We call such edges *critical*:

Definition 3. An edge $(s_1, l, s_2) \in E_{\mathcal{G}'}$ is called *critical* with respect to a graph extension $\mathcal{G} \sqsubseteq \mathcal{G}'$ and a query $s \xrightarrow{r}\!\!\!\!\!\rightarrow D$ if there exist paths p_1 and p_2 that satisfy the following conditions:

(a) $\mathcal{G} \vdash p_1 : s \xrightarrow{w_1} s_1$ for some word w_1,

(b) $\mathcal{G}' \vdash p_2 : s_2 \xrightarrow{w_2} s_3$ for some node s_3 and word w_2,

(c) $(p_1 \cdot l \cdot p_2) \in \mathsf{Ans}\left(\mathcal{G}', s_1 \xrightarrow{r}\!\!\!\!\!\rightarrow D\right)$,

(d) and $(s_1, l, s_2) \notin E_{\mathcal{G}}$.

Fig. 4.10 visualizes the critical edges for a particular graph extension and query. Critical edges for a query are interesting because *their absence* in a graph extension guarantees stability of the answer to that query:

Lemma 2 (Absence of Critical Edges). *A reachability query $s \xrightarrow{r}\!\!\!\!\!\rightarrow D$ is stable under graph extension $\mathcal{G} \sqsubseteq \mathcal{G}'$ iff $\mathcal{G} \sqsubseteq \mathcal{G}'$ contains no critical edges for $s \xrightarrow{r}\!\!\!\!\!\rightarrow D$.*

Proof sketch. This absence of critical edges implies stability because *every* path that answers a query that is in the extended graph $\mathcal{G}'$ *but not* in the original graph $\mathcal{G}$ can be partitioned as $p_1 \cdot l \cdot p_2$ such that $(\mathsf{tgt}\,(p_1)\,, l, \mathsf{src}\,(p_2))$ is a critical edge. Consequently, the absence of critical edges in an extension immediately implies that the extended graph yields no new answers to the query under scrutiny. The other direction of this lemma holds trivially. □

As indicated by Lemma 2, it would be sufficient for the rule for queries to require the absence of critical edges in future graphs. The problematic question however is: critical with respect to which graph extension? Indeed, the graphs $\mathcal{G}'$ that Lemma 2 quantifies over are *all future graphs* of a trace in the operational semantics. Precisely knowing $\mathcal{G}'$ is as difficult as solving the constraint program. Hence it is not feasible for a solver to guard against the absence of critical edges with pinpoint accuracy. In the remainder of this section we describe a two-part approach to sound operation of a non-backtracking solver based on *over-approximating* the criticality of an edge.

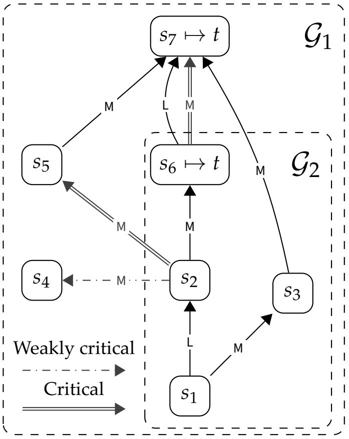


Figure 4.10: (Weakly) critical edges for the query $s_1 \overset{LM^*}{\twoheadrightarrow} D$ (if $t \in D$).

Weakly critical edges Because the notion of criticality is derived from entire new reaching paths in graph extensions, guarding against critical edge extensions requires looking ahead over arbitrary constraint solving. Our approximation, a *weakly critical* edge, reduces the required lookahead to just one-edge extensions of the *current* graph:

Definition 4. An edge (s_1, l, s_2) is called *weakly critical* with respect to a graph $\mathcal{G}$ and a query $s \overset{r}{\twoheadrightarrow} D$ if there exists a path p_1 that satisfies the following conditions:

(a) $\mathcal{G} \vdash p_1 : s \xrightarrow{w_1} s_1$ for some word w_1,
(b) the word $(w_1 l)$ is a prefix of some word in $\mathcal{L}(r)$,
(c) and $(s_1, l, s_2) \notin E_{\mathcal{G}}$.

In Fig. 4.10 an edge is highlighted that is only weakly critical: it shares all the features of a critical edge except that it does not actually give rise to new paths in the answer set of the query. The intuition behind a weakly critical edge is that it *may* lead to additional reaching paths. Every critical edge is also weakly critical, such that the following corollary holds:

Corollary 1. *A reachability query $Q = s \overset{r}{\twoheadrightarrow} D$ is stable under graph extension $\mathcal{G}_1 \sqsubseteq \mathcal{G}_2$ if the graph extension $\mathcal{G}_1 \sqsubseteq \mathcal{G}_2$ contains no edges that are weakly critical for Q.*

Proof sketch. Every critical edge is also weakly critical because

$$(p_1 \cdot l \cdot p_2) \in \mathsf{Ans}\left(\mathcal{G}', s_1 \overset{r}{\twoheadrightarrow} D\right)$$

implies that (wl) is a prefix of some word in $\mathcal{L}(r)$, for $w = \mathsf{labels}\,(p_1)$. The conclusion then immediately follows from Lemma 2. □

Because visibility is defined as the minimum of a reachability query answer (§4.3.1), the absence of weakly critical edges is also a *sufficient condition* for stability of visibility query answers.

Corollary 2 (Absence of Weakly Critical Edges). *A visibility query $Q = s \overset{r}{\twoheadrightarrow} D$ is stable under graph extension $\mathcal{G}_1 \sqsubseteq \mathcal{G}_2$ if the graph extension $\mathcal{G}_1 \sqsubseteq \mathcal{G}_2$ contains no edges that are weakly critical for Q.*

Consequently, the absence of weakly critical edges is also sufficient to guarantee the soundness of visibility queries with *any* path order $\leq_{\mathsf{p}}$. However, for particular choices of the path order there exist tractable approximations of criticality of edges for stability of reachability that are more precise than weak criticality. For example, the path ordering is often defined as the lexicographical extension of a precedence ordering on edge labels. Edge extensions of the graph with lower precedence than existing edges can in that case be disregarded as influential to name resolution. Our results extend to such refinements in a straightforward manner.

4.5.3 Guarded Query Answering

By means of a well-formedness judgment $\vdash C$ wf on Statix-core constraints, we define a large class of constraints for which we can check the absence of weakly critical edges. To this end we will also define a predicate $C \not\rightarrowtail (s, l)$ which can be checked syntactically, but has the semantics that C does not support any l-edges out of s *if* C is well-formed. We then prove the following *guarded query simplification rule* correct:

Op-Query-Guarded

$$\frac{\forall s_2, l.\ \left(\mathcal{G} \vdash p : s_1 \xrightarrow{w} s_2 \text{ and } \mathcal{L}(\delta_{wl} r) \neq \emptyset \text{ implies } (C; \overline{C}) \not\rightarrowtail (s_2, l)\right)}{\left\langle \mathcal{G} \mid \mathsf{query}\ s_1 \overset{r}{\twoheadrightarrow} D\ \mathsf{as}\ z.C; \overline{C} \right\rangle \to \left\langle \mathcal{G} \mid C\left[\mathsf{Ans}\left(\mathcal{G}, s_1 \overset{r}{\twoheadrightarrow} D\right)/z\right]; \overline{C} \right\rangle}$$

Recall that the $\mathcal{L}(\delta_{wl} r) \neq \emptyset$ denotes that (wl) is a prefix of some word in $\mathcal{L}(r)$. Intuitively, the precondition states that the remainder of the constraint program does not support any weakly critical edges for the query under scrutiny.

Well-formed constraints. We define well-formedness inductively using the rules in Fig. 4.11. The intuition behind well-formed constraints is that asserting new outgoing edges on nodes requires *permission to extend* that scope. This judgment is defined in terms of an auxiliary judgment $\Delta^{\downarrow}, \Delta^{\uparrow} \vdash C$ which denotes that the constraint C requires permission for variables in $\Delta^{\downarrow}$, and has permission for those in $\Delta^{\uparrow}$.

Syntactical extends predicate. We also inductively define a syntactical judgment $C \hookrightarrow (s, l)$ in Fig. 4.11, denoting that C supports an edge (s, l, s') for some s'. We write $C \not\hookrightarrow (s, l)$ to denote its negation. We lift both relations to work on constraint *sets*. The key result is the following:

Lemma 3. *For all well-formed constraints the syntactical approximation of absence of support implies the semantic counterpart. That is:*

$$\frac{\vdash C \text{ wf} \qquad C \not\hookrightarrow (s,l) \qquad \mathcal{G}, \varphi \vDash_{\sigma} C \qquad s \notin \sigma}{\forall s'.\, (s,l,s') \notin \sigma}$$

Proof sketch. We prove a stronger property, whose assumptions hold under the premises of the lemma in question:

$$\frac{\Delta^{\downarrow}, \Delta^{\uparrow} \vdash C \qquad \Big(\forall (x \in \Delta^{\downarrow}) \Rightarrow (x\varphi \neq s)\Big) \qquad C \not\hookrightarrow (s,l) \qquad \mathcal{G}, \varphi \vDash_{\sigma} C \qquad s \notin \sigma}{\forall s'.(s,l,s') \notin \sigma}$$

The proof itself is by induction on C. The interesting case to consider is edge assertions. In case the source of the edge is ground, the conclusion follows from inversion of the third premise $(s' \xrightarrow{l'} t) \not\hookrightarrow (s, l)$. In case the source of the edge is represented by a variable x, the first premise guarantees $x \in \Delta^{\downarrow}$, such that the conclusion follows by the second premise. □

Equally important is the fact that $\vdash C$ wf is preserved by steps. That allows it to be checked only once on the input program without dynamically enforcing it on intermediate constraint sets.

Theorem 3. *Steps preserve well-formedness of constraints:*

$$\big(\langle \mathcal{G} \mid \overline{C}_1 \rangle \rightarrow \langle \mathcal{G}' \mid \overline{C}_2 \rangle \textit{ and } \vdash \overline{C}_1 \text{ wf}\big) \textit{ imply } \vdash \overline{C}_2 \text{ wf}$$

Using the fact that absence of weakly critical edges is sufficient for stability (Lemma 1), *and* the fact that the absence of weakly critical edges can be ensured for well-formed constraints (Lemma 3), we prove that the guarded simplification rule preserves satisfiability of the constraint problem:

$\vdash C\ \mathsf{wf}$ Constraint program C has sufficient permissions

$$\frac{\Delta^{\downarrow}, \Delta^{\uparrow} \vdash C \qquad \Delta^{\downarrow} \subseteq \Delta^{\uparrow}}{\vdash C\ \mathsf{wf}}\ \textsc{Wf-Program}$$

$\Delta^{\downarrow}, \Delta^{\uparrow} \vdash C$ C requires permissions for names in $\Delta^{\downarrow}$ and provides them for $\Delta^{\uparrow}$

$$\textsc{Wf-True}\quad \frac{}{\emptyset, \emptyset \vdash \mathsf{emp}}$$

$$\textsc{Wf-False}\quad \frac{}{\emptyset, \emptyset \vdash \mathsf{false}}$$

$$\textsc{Wf-Conj}\quad \frac{\Delta_1^{\downarrow}, \Delta_1^{\uparrow} \vdash C_1 \qquad \Delta_2^{\downarrow}, \Delta_2^{\uparrow} \vdash C_2}{\Delta_1^{\downarrow} \cup \Delta_2^{\downarrow}, \Delta_1^{\uparrow} \cup \Delta_2^{\uparrow} \vdash C_1 * C_2}$$

$$\textsc{Wf-Eq}\quad \frac{}{\emptyset, \emptyset \vdash t_1 = t_2}$$

$$\textsc{Wf-Exists}\quad \frac{\Delta^{\downarrow}, \Delta^{\uparrow} \vdash C \qquad \left(x \in \Delta^{\downarrow} \Rightarrow x \in \Delta^{\uparrow}\right)}{\Delta^{\downarrow} \setminus \{x\}, \Delta^{\uparrow} \setminus \{x\} \vdash \exists x.C}$$

$$\textsc{Wf-Singleton}\quad \frac{}{\emptyset, \emptyset \vdash \mathsf{single}(t, \bar{t})}$$

$$\textsc{Wf-Node-Var}\quad \frac{}{\emptyset, \{x\} \vdash \nabla x \mapsto t}$$

$$\textsc{Wf-Node-NoVar}\quad \frac{t \text{ is not a variable}}{\emptyset, \emptyset \vdash \nabla t \mapsto t'}$$

$$\textsc{Wf-Forall}\quad \frac{\Delta^{\downarrow}, \Delta^{\uparrow} \vdash C \qquad \left(x \in \Delta^{\downarrow} \Rightarrow x \in \Delta^{\uparrow}\right)}{\Delta^{\downarrow} \setminus \Delta^{\uparrow} \setminus \{x\}, \emptyset \vdash \forall x\ \mathsf{in}\ \bar{t}.C}$$

$$\textsc{Wf-Edge-Var}\quad \frac{}{\{x\}, \emptyset \vdash x \xrightarrow{l} t'}$$

$$\textsc{Wf-Edge-NoVar}\quad \frac{t \text{ is not a variable}}{\emptyset, \emptyset \vdash t \xrightarrow{l} t'}$$

$$\textsc{Wf-Query}\quad \frac{\Delta^{\downarrow}, \Delta^{\uparrow} \vdash C}{\Delta^{\downarrow}, \Delta^{\uparrow} \vdash \mathsf{query}\ s \overset{r}{\twoheadrightarrow} D\ \mathsf{as}\ z.C}$$

$$\textsc{Wf-Data}\quad \frac{}{\emptyset, \emptyset \vdash \mathsf{dataOf}(t, t')}$$

$C \hookrightarrow (s, l)$ Constraint program C asserts an l-edge on node s

$$\textsc{Ext-Conj1}\quad \frac{C_1 \hookrightarrow (s, l)}{(C_1 * C_2) \hookrightarrow (s, l)}$$

$$\textsc{Ext-Conj2}\quad \frac{C_2 \hookrightarrow (s, l)}{(C_1 * C_2) \hookrightarrow (s, l)}$$

$$\textsc{Ext-Exist}\quad \frac{C \hookrightarrow (s, l)}{(\exists x.C) \hookrightarrow (s, l)}$$

$$\textsc{Ext-Edge}\quad \frac{}{(s \xrightarrow{l} t) \hookrightarrow (s, l)}$$

$$\textsc{Ext-Forall}\quad \frac{C \hookrightarrow (s, l)}{(\forall \bar{t}\ \mathsf{in}\ z.C) \hookrightarrow (s, l)}$$

$$\textsc{Ext-Query}\quad \frac{C \hookrightarrow (s, l)}{(\mathsf{query}\ t \overset{r}{\twoheadrightarrow} D\ \mathsf{as}\ z.C) \hookrightarrow (s, l)}$$

Figure 4.11: Well-formed constraints and syntax directed edge support predicate.

Theorem 4. *The guarded simplification step preserves satisfiability:*

$$\frac{\left(\forall s_2, l . \mathcal{G} \vdash p : s_1 \overset{w}{\rightarrow} s_2 \textit{ and } \mathcal{L}(\delta_{wl} r) \neq \emptyset \textit{ imply } (\mathsf{C}; \overline{\mathsf{C}}) \not\rightarrow (s_2, l)\right)}{[\![\mathcal{G}]\!] \ast \mathsf{query}\ s_1 \overset{r}{\twoheadrightarrow} D\ \mathsf{as}\ z.C \ast (\mathop{\ast} \overline{\mathsf{C}}) \mathrel{\sim\!\dashv\!\vdash\!\sim} [\![\mathcal{G}]\!] \ast \mathsf{C}\left[\mathsf{Ans}\left(\mathcal{G}, s_1 \overset{r}{\twoheadrightarrow} D\right)/z\right] \ast (\mathop{\ast} \overline{\mathsf{C}})}$$

Proof sketch. We prove this equivalence in the direction right to left. The other direction proceeds similarly. The hypothesis states that there is a graph $\mathcal{G}'$, which is a supported model for the right hand side of the above equivalence:

$$\mathcal{G}', \varphi \models [\![\mathcal{G}]\!] \ast \mathsf{C}\left[\mathsf{Ans}\left(\mathcal{G}, s_1 \overset{r}{\twoheadrightarrow} D\right)/z\right] \ast (\mathop{\ast} \overline{\mathsf{C}}) \tag{I}$$

We prove that the substituted answer to the query is stable for the extension $\mathcal{G} \sqsubseteq \mathcal{G}'$. The conjunction distributes support in disjoint fashion over the operands, and the embedding of $\mathcal{G}$ requires support for all of its nodes and edges. Consequently:

$$\mathcal{G}', \varphi \models_{\langle S_{\mathcal{G}'} \setminus S_{\mathcal{G}}, E_{\mathcal{G}'} \setminus E_{\mathcal{G}} \rangle} \mathsf{C}\left[\mathsf{Ans}\left(\mathcal{G}, s_1 \overset{r}{\twoheadrightarrow} D\right)/z\right] \ast (\mathop{\ast} \overline{\mathsf{C}}) \tag{II}$$

Now assume a weakly critical edge (s_2, l, s_3). By definition we must have that $\mathcal{G} \vdash p : s_1 \overset{w}{\rightarrow} s_2$ and $\mathcal{L}(\delta_{wl} r) \neq \emptyset$. From the guard of the query simplification rule we may conclude $(\mathsf{C}; \overline{\mathsf{C}}) \not\rightarrow (s_2, l)$. This relation is preserved under the answer set substitution into the constraint C. Lemma 3 now ensures that the remainder of the constraint program cannot support the weakly critical edge:

$$\forall s_3 . (s_2, l, s_3) \notin (E_{\mathcal{G}'} \setminus E_{\mathcal{G}})$$

It follows by Lemma 1 that the answer set is stable for this graph extension:

$$\mathsf{Ans}\left(\mathcal{G}, s_1 \overset{r}{\twoheadrightarrow} D\right) = \mathsf{Ans}\left(\mathcal{G}', s_1 \overset{r}{\twoheadrightarrow} D\right) \tag{III}$$

Combining (I) and (III), we have:

$$\mathcal{G}', \varphi \models [\![\mathcal{G}]\!] \ast \mathsf{C}\left[\mathsf{Ans}\left(\mathcal{G}', s_1 \overset{r}{\twoheadrightarrow} D\right)/z\right] \ast (\mathop{\ast} \overline{\mathsf{C}})$$

The desired result follows from query-introduction in the middle operand. □

We have proven that all steps in the extended operational semantics preserve satisfiability. Soundness follows:

Theorem 5 (Soundness of Statix-Core with Queries). *If the operational semantics accepts a closed and* well-formed *constraint C, i.e. $\langle \epsilon \mid \{C\} \rangle \rightarrow^* \langle \mathcal{G} \mid \emptyset \rangle$, then the resulting graph is a supported model for that constraint: $\mathcal{G} \models C$. If C is rejected, then no supported model exists.*

Proof sketch. The proof is the same as the proof for soundness of the fragment without queries, using Thm. 4 to prove that the additional step in the operational semantics also preserves satisfiability. □

We end our discussion of the extended operational semantics by observing that it is still confluent. The interesting critical pair reduces a query in the left step and an edge asserting in the right step. The diamond is formed using the fact that the premise of the query step ensures that the asserted edge cannot be critical for the query Rouvoet, van Antwerpen, et al., 2020c, Appendix A.

4.6 Implementation and Case Studies

We developed the operational semantics of Statix-core and have proven that the operational semantics computes sound name resolution results for well-formed specifications. However, the well-formedness restriction and the possibility that the scheduling gets stuck limits the expressiveness of Statix-core. In this section we describe an evaluation of our approach using MiniStatix: a prototype implementation of Statix that closely follows the operational semantics.

MiniStatix implements the core constraint language Statix-core, as well as (mutually) recursive predicates and (guarded) pattern matching, in approximately 3000 lines of Haskell. The language has a simple module system to enable the larger case study language specifications to be organized across files. After parsing, the specification is statically checked: names are statically resolved, after which *permissions are inferred* for constraints, deriving the relation formally stated in Fig. 4.11. The implementation extends the definition of permissions and well-formedness to predicates and pattern matching.

The solver implementation is a variation of the small-step operational semantics that uses environments rather than substitution. It uses a round-robin, delaying scheduler for constraints, which can detect configurations where no more progress can be made (i.e., stuckness). For satisfied constraints, the solver outputs a complete scope graph and the unifier for the top-level existential quantifier if there is any. For rejected programs, the solver will give the trace of instantiated predicates that led to falsification, which functions as a formal explanation of the error. Stuck configurations are output for specification debugging purposes.

We have evaluated our approach using MiniStatix on three case studies by implementing a subset of name resolution for Java and Scala, and the whole of LMR (van Antwerpen et al., 2016): a toy language with modules and records. The former two show that our approach can indeed resolve challenging patterns of real languages. By targeting subsets of real languages, we are able to directly test our approach against the Java and Scala type checker. The test succeeds if MiniStatix and the reference type checker agree on whether a test program is valid. Programs that should be

Table 4.1: Evaluation: test results.

Language	LOC Spec	Tests	Succeed	Fail	Stuck
Java	1201	125	125	0	0
Scala	517	109	109	0	0
LMR	263	19	15	0	4
Total	1976	253	249	0	4

rejected are equipped with specific error expectations to avoid false positives. The third case study (LMR) is used to explain when our approach is incomplete, causing stuck configurations in MiniStatix. We count a test case as a success if it does not get stuck and meets the manually set test expectation (because LMR has no reference type checker). The results are summarized in Table 4.1 and we briefly highlight some parts of the case studies below.

The implementation of MiniStatix, the language specifications and tests are available as an artifact accompanying this paper (Rouvoet, van Antwerpen, et al., 2020b).

The Java study We selected a subset of Java with a focus on the binding aspects of packages, imports, classes, interfaces, inheritance, inner classes, and method and field members. Test cases are set up so that faulty name resolutions result in type errors and focus on interesting edge cases. The tests come in pairs that test that good programs are accepted and ill-typed variants are rejected.

Packages in Java are an interesting test subject because at first sight they seem to require *remote extension*—i.e. the very pattern that is forbidden by our well-formedness restriction. Package names in Java have no authoritative declaration, but exist by virtue of use. More than one compilation unit can declare to define members in the same package. The well-formedness restriction indeed does not permit modeling this by resolving the package name at the top of a compilation unit to obtain a package scope and contributing definitions to that scope. This would constitute *remote extension* of the package scope. However, the right binding semantics can also be modeled via a mixin-pattern: compilation units query for all other compilation units in the same package and make their types accessible by adding import edges. This model makes it locally very apparent what things are in scope of the compilation unit, and also passes the well-formedness check so that stability of query answers can be guaranteed.

The Scala study The focus of the Scala case study is resolution of names to local definitions and imports. Scala not only gives different precedence levels to local definitions, wildcard, and specific imports, but also distinguishes their scope. Concretely, local definitions are accessible in the *surrounding* scope to accommodate mutual definitions, whereas imported names are only accessible in *subsequent* scope. This en-

```
object c {
  import a._;
  def g(): Unit = {
    val x: Int = h();
    import b.h;
  };
  def h(): Int = 42;
};
object a {
  object b {
    def h(): Unit = {};
  };
};
```

(a) Scala's scoping.

```
pub mod foo {
  pub mod bar {}
}

pub mod test {
  use super::*;
  use bar::*;
  use foo::*;
}
```

(b) Well-typed Rust example.

```
pub mod foo {
  pub mod foo {}
}

pub mod test {
  use super::*;
  use foo::*;

}
```

(c) Ambiguous Rust example.

Figure 4.12: Example programs from the case studies.

sures that resolving import statements cannot influence their own resolution. This simplifies scheduling because it avoids the need to iterate name resolution within a block. We discuss iterated name resolution (which Rust and LMR require) in more detail below.

The well-typed example test case in Fig. 4.12a highlights the scoping difference between declarations and imports, and also shows specific imports, wildcard imports, and imports from imported objects. The forward reference to the locally defined object `a` is well bound, whereas the imported definition of `h` cannot be forward referenced. In addition to the shown features, our Scala subset supports hiding and renaming in imports.

The LMR/Rust study As a third study we looked at a language that has imports that can affect their own resolution. (An extended version of the following discussion can be found in Rouvoet, van Antwerpen, et al., 2020c, Appendix C). Although this does not appear to be a common language feature, at least Rust does implement this import semantics. The difficulty arises because LMR and Rust combine features that are not usually found together in other module systems: 1. relative imports, 2. unordered imports, and 3. glob imports. The combination of these features make programs as depicted in Fig. 4.12b well-typed. In contrast, Scala has imports that resolve relative to the local scope, but they only open in subsequent scope—i.e., they are *ordered*. The direct Scala equivalent of the given example would therefore not be able to resolve the name `bar`. Example Fig. 4.12c shows how this can lead to strange name resolution situations where imports are self-influencing. The Rust type checker judges this program to be ambiguous: imports do not shadow outer declarations, so that two declarations of `foo` are visible in the block of module `test`.

The Rust type checker uses iterated name resolution to implement the desired behavior, re-resolving module names until the environment stabilizes. MiniStatix on the other hand gets stuck on Rust/LMR programs with imports—i.e., also non-ambiguous programs. The import is specified using a query and an import edge assertion. However, the query is delayed on the weakly critical edge assertion that in turn is waiting on the query to resolve the target scope of the edge.

The difference between Scala's and Rust's imports exactly exposes the limits of our particular over-approximation of dependencies using *weakly* critical edges: it may lead to the operational semantics being stuck on programs that in principle have a stable model. Rust shows that a sound fixed point algorithm exists for name resolution in Rust programs. How to systematically derive such an algorithm from high-level declarative specifications is a different question. From a declarative specification of self-influencing imports some paradoxes can arise. It is worth pondering what should be the meaning of Fig. 4.12c if imports *do* shadow outer declarations.

4.7 Related Work

The main novelties of the Statix specification language compared to typical typing rules are the assertions of scope graph structure, and the queries over the resulting graph. The fact that scopes are passed by reference enables the high-level specification of name binding in two ways. First, it makes it possible to separate the assertion that a scope exists from the description of its contents. This is useful because scope is naturally a concept that extends over larger parts of syntax, whereas typing rules are usually given by induction over the syntax. Second, it makes retrieving binding information about remote parts of the AST lightweight because it is accessible via scope references. This makes it unnecessary to propagate and construct complicated environments in typing rules.

At the same time, these features present a challenge operationally. In order to maintain soundness with respect to the declarative semantics, queries need to be delayed until all contributions to the relevant scopes have been witnessed. This paper addresses that challenge. In this section, we want to relate to and compare with other approaches to operationalizing declarative specifications of static semantics.

4.7.1 Constraint Generation and Solving

Statix is a constraint language in the tradition of Constraint Handling Rules (CHR; T. W. Frühwirth, 1998). CHR has a sound semantics of fact assertion and retraction. Fact assertion and retraction are considered impure primitives in Prolog (Moss, 1986). Where CHR uses the constraint store to record assertions, Statix uses the scope graph.

Unlike constraint store facts, scope graph facts are only asserted and never retracted. The context-sensitive effects that can be achieved using multi-head propagation and simpagation rules in CHR can be realized using scope graph constraints in Statix.

The approach of CHR and Statix is distinctly different from approaches that separate the constraint generation and constraint solving phases in the tradition of Hindley-Milner type-inference (Odersky et al., 1999; Pottier and Rémy, 2005). The constraint-generation based formalism that is closest to Statix is its precursor NaBL2 (van Antwerpen et al., 2016). Like Statix, it has built-in support for name resolution using scope graphs (Néron et al., 2015), but separates constraint generation from constraint solving.

NaBL2 supports type-dependent name resolution, in which the resolution of a name (such as the method name in $e.m()$) depends on the resolution of a type (for the receiver expression e), which in turn may depend on name resolution. It has to deal with the fact that sometimes not all binding information is available when a name is resolved. The incomplete information is represented explicitly in the model using an *incomplete scope graph*, where unification variables can be placeholders for scopes. During constraint solving, such unification variables must be unified before they can be traversed as part of queries. The solver guarantees query stability by relying on a resolution algorithm that *delays* when resolution encounters an edge to a unification variable.

Unlike in Statix, scope graphs in NaBL2 can *only* be incomplete in the sense that the target of an edge is yet unknown. Edges cannot be missing entirely. This prohibits specifications where the presence of edges is dependent on resolution in the scope graph. In Statix this is permitted and used (van Antwerpen et al., 2018). For example, imports-with-hiding in our Scala case study is specified using a query that finds all members of an object scope and a new scope that is a masked version of the object scope. The number of edges of the masked scope depends on query resolution.

4.7.2 On-demand Evaluation of Canonical Attribute Grammars

Another way to operationalize a type system is to use an *attribute grammar* (AG), using equations on AST nodes to define the values of *attributes*. Attributes are either inherited (i.e., computed by the parent and propagated down the AST), or synthesized (i.e., computed on the node itself and propagated upwards). Name resolution can be specified using AGs by taking environment-based typing rules such as in Fig. 4.2b and turning the downwards and upwards propagating environments into inherited and synthesized attributes respectively.

Canonical attribute grammars were implemented by statically computing a schedule (or plan) consisting of multiple passes over the AST, ordered such that the input values of the attribute computations in one pass are computed in a previous pass (for a survey, see Alblas, 1991). Expressivity of canonical attribute grammars is limited by this stratified evaluation. By building on the circular programming techniques of Bird (1984), Johnsson (1987) shows how dependencies between attributes can be determined dynamically, relaxing the non-circularity requirements on specifications. Modern attribute grammar formalisms like JastAdd (Ekman and Hedin, 2006, 2007a) and Silver (Van Wyk et al., 2010) use these techniques, relying mostly on on-demand computation.

The *specification* problems that we describe in §4.2 with environment-based rules also affect canonical attribute grammars (AG). In particular, to gain access to binding information from somewhere else in the tree, this information needs to be aggregated and distributed through the least common ancestor (J. T. Boyland, 2005). This leads to more complex, non-modular grammars for languages with complex binding rules (Hedin, 2000). This specification problem is the motivation for *Reference Attribute Grammars* (RAGs), which we discuss separately below.

J. T. Boyland (2005) also describes how canonical AGs suffer from an *implementation* problem: packaging multiple values into environment attributes requires that they can be computed at the same time. Sometimes this causes circular dependencies that disappear when values are split across multiple environments. This means that the specification writer has to be aware of the operational semantics. J. T. Boyland (2005) concludes: "The decision of whether two values can be packaged together (thus reducing complexity and increasing efficiency) relies on global scheduling information, and thus should be left to an implementation tool, not the description writer." This motivates the development of *Remote Attribute Grammars*. The same problem also motivated the design of Statix.

4.7.3 Scheduling of Reference Attribute Grammars with Collection Attributes

Reference attributes (Hedin, 2000) are an extension of canonical AGs that allow attributes that *reference* AST nodes. Attributes of the referenced AST nodes can be read directly. This can be used to avoid the need to propagate information using environments, and thus avoids some of the problems with the specification and the implementation of static semantics using environments that we described in §4.2. Reference attributes can be used to superimpose graphs on an AST.

By themselves, reference attributes do not solve the problems with the aggregation of binding described in §4.2. To additionally avoid the specification overhead of aggregating values from an AST, they can be combined with parameterized attributes or collection attributes (J. Boyland, 1996).

Parameterized attributes are used for example to define name resolution for large subsets of Java in the JastAdd AG system (Ekman and Hedin, 2007a,b). This is accomplished by defining a parameterized `lookup` attributes on nodes that implement the name resolution policy. These attributes are invoked on references, passing the name to be resolved. Shadowing can be implemented by deferring to the lookup of child and parent nodes in a particular order. The effective resolution policy for the resolution of a variable is thus determined by the combination of all local policies implemented in the nodes that are traversed. This differs significantly from Statix specifications, where the resolution policy is determined more uniformly by the query parameters in the variable rule. The separation queries from scope graph construction in Statix is designed to make it easy to extract an abstract model of binding. Parameterized attributes are evaluated on-demand.

Collection attributes collect *contributions* that can come from different contributor nodes throughout the AST. A contributor uses a reference attribute to specify to which collection it contributes. The mutual binding example in Fig. 4.2a can be specified using reference and collection attributes. A block defines a collection attribute that collects the binding contributions from its immediate children. To that end the children need a reference to the block, which can be specified as an inherited attribute. We are not aware of any case studies involving non-lexical static binding that make use of collection attributes for name resolution.

There are two approaches to evaluating AGs with collection attributes. The first approach is due to Magnusson et al. (2009). Before a collection is read, all contributions must have been computed. To be able to determine if this is the case, a pass is made over the AST and for all contributions to *any* instance of the collection attribute, the reference that is contributed to it is evaluated. Like in Statix, this is an over-approximation of dependencies. After this, all contributions are evaluated for the one reference whose collection is being read. Because of the first pass, the reference attribute can never depend on any instance of the collection attribute, or a cycle would occur (Magnusson et al., 2009). This can cause evaluation to get stuck even when sound schedules exist.

The specification of contributions differs from the specification of edges in Statix, in that edge assertions can occur anywhere in a specification on any scope reference. In a Statix specification that does not enforce our permission-to-extend restriction, it is *not* possible to demand the evaluation of the scope reference that the edge is 'contributed to'. This is the case because the scope reference can be determined by arbitrary constraints, which can be blocked. On the other hand, if permission-to-extend *is* enforced, then it is unnecessary to evaluate all scope references that are contributed to. This is the case because a scope that is not yet ground cannot be instantiated to any already existing scope—hence $(x \xrightarrow{l} t) \not\rightarrowtriangle (s, l)$ is sound.

A Statix specification has no immediate counterpart as a RAG. An obstacle is that Statix rules do not clearly distinguish inputs and outputs, which is part of their declarative appeal. It also potentially enables them to be used to solve other language implementation problems that involve the static semantics, such as suggesting well-typed program completions (Daniel A. A. Pelsmaeker et al., 2019). Attribute grammars on the other hand organize specifications into equations for attributes, which have a clear direction. A benefit of this approach is that dependencies are more explicitly present in the specification (even for equations that specify contributions to collection attributes), so that on-demand evaluation is available. Encoding Statix rules into AG equations requires a factorization into attributes. Whether this is always possible is an interesting open research problem.

4.8 Conclusion

We envision closing the gap between language specification and language implementation by using specification languages that can address the complexity of actual programming languages and systematically deriving implementations from specifications. Importantly, this moves the question of implementation correctness from the concrete language to the specification language. This approach leads to correct-by-construction language implementations and higher-level specifications that abstract from operational concerns.

In this paper, we tackled one aspect of that challenge. Critical edges represent language independent insight into a scheduling problem that type checker implementations need to address. Because it is a high-level concept, it can be used to think about language design. We exploit this insight and obtain sound-by-construction scheduling in type checkers derived from specifications.

Interesting future research topics are the declarative specification of dependently typed languages, and type inference beyond what is covered by Statix's support for unification. It would also be interesting to investigate support for user-defined fixed point properties (Magnusson and Hedin, 2003; Sasaki and Sassa, 2003), enabling the specification of data-flow analyses in Statix.

Acknowledgments We thank Andrew Tolmach and Reuben Rowe for the many discussions on scope graphs and constraint semantics and resolution. We also thank the anonymous reviewers of this paper and its prior submissions for their feedback and valuable suggestions. This research was funded by the NWO VICI Language Designer's Workbench project (NWO 639.023.206), the NWO VENI Verified Programming Language Interaction project (NWO 016.Veni.192.259), and the NWO VENI Composable and Safe-by-Construction Programming Language Definitions project (NWO VI.Veni.192.259).

Scope States

5

Abstract Compilers that can type check compilation units in parallel can make more efficient use of multi-core architectures, which are nowadays widespread. Developing parallel type checker implementations is complicated by the need to handle concurrency and synchronization of parallel compilation units. Dependencies between compilation units are induced by name resolution, and a parallel type checker needs to ensure that units have defined all relevant names before other units do a lookup. Mutually recursive references and implicitly discovered dependencies between compilation units preclude determining a static compilation order for many programming languages.

In this paper, we present a new framework for implementing hierarchical type checkers that provides implicit parallel execution in the presence of dynamic and mutual dependencies between compilation units. The resulting type checkers can be written without explicit handling of communication or synchronization between different compilation units. We achieve this by providing type checkers with an API for name resolution based on scope graphs, a language-independent formalism that supports a wide range of binding patterns. We introduce the notion of *scope state* to ensure safe name resolution. Scope state tracks the completeness of a scope, and is used to decide whether a scope graph query between compilation units must be delayed. Our framework is implemented in Java using the actor paradigm. We evaluated our approach by parallelizing the solver for Statix, a meta-language for type checkers based on scope graphs, using our framework. This parallelizes every Statix-based type checker, provided its specification follows a split declaration-type style. Benchmarks show that the approach results in speedups for the parallel Statix solver of up to 5.0x on 8 cores for real-world code bases.

5.1 Introduction

Despite the general availability of multi-core architectures, many compilers do not take advantage of these for type checking. Parallelizing a compiler remains a challenging task, which requires dealing with explicit synchronization and communication between compilation units. For example, the authors of GCC made the following remark about their efforts to parallelize parts of the compiler (GCC, n.d.):

> *"One of the most tedious parts of the job was [...] making several global variables threadsafe, and they were the cause of most crashes in this project."*

Published as Hendrik van Antwerpen and Eelco Visser (2021). "Scope States: Guarding Safety of Name Resolution in Parallel Type Checkers." In: *35th European Conference on Object-Oriented Programming (ECOOP 2021)*. Dagstuhl, Germany. DOI: 10.4230/LIPIcs.ECOOP.2021.1.

They continue to say that even with the help of specialized tools it remained difficult to do the parallelization correctly. Compilers for many major languages do not support parallel front ends, or only experimentally. Some build tools allow the compiler to be run in parallel, but many require a static compilation order, and because they have no internal knowledge of the language being compiled, cannot generically handle cyclic dependencies between compilation units. A generic, reusable solution to the problem of how to implement type checkers that can process compilation units in parallel, which correctly deals with (potentially cyclic) dependencies between units, is missing[1].

Dependencies between compilation units are the results of name lookup from one unit into another. A correct concurrent type checker must ensure that when a lookup is done, all relevant units have progressed enough to provide a complete answer. For languages that support true separate compilation (e.g., Shao and Appel, 1993), there are no lookups into other units, and processing them in parallel is trivial. It may also be possible to run type checkers in parallel using a compilation order based on static or dynamic dependencies, which ensures units are compiled after their dependencies. But many programming languages have features, such as mutually recursive modules, that result in mutual dependencies between compilation units. When compilation units are mutually dependent, neither unit can be completed before the other is at least partially checked. A more fine-grained approach than processing compilation units in a fixed order is required.

This paper presents a new framework for the implementation of type checkers that provides *implicit parallel execution*. Type checkers are organized as a hierarchy of compilation units, which allows modeling simple scenarios such as flat files in a project, as well as package hierarchies. The framework supports *dynamic dependencies* and *mutual dependencies* between compilation units. The type checkers can be written without the need to explicitly handle communication or synchronization between units.

This is achieved by providing type checkers with an API for name resolution based on scope graphs. Scope graphs are a language-independent formalism for name binding and name resolution, which has been shown to support a wide range of binding patterns, and has successfully been applied to implement type checkers (Néron et al., 2015; Rouvoet, van Antwerpen, et al., 2020a; van Antwerpen et al., 2018). The key to our approach is twofold:

- delay lookups when other units have not progressed enough to give a *safe*, that is, complete answer, and
- release delayed queries as soon as possible, even if other parts of the graph are still incomplete.

[1] The type checkers we envision are both concurrent (i.e., units make (interleaved) progress during the same period) and parallel (i.e., units run at the same time), and we use the terms interchangeably.

Recent work by Rouvoet, van Antwerpen, et al. (2020a) identifies the absence of *weakly critical edges* as a sufficient condition to guarantee safe name resolution in a partial scope graph. We develop the notion of *scope state* to allow fine-grained tracking of the presence or absence of weakly critical edges. Through these scope states, which are managed by the type checkers via the name resolution API, the framework ensures safety of name resolution. The provided API is asynchronous, which works with type checkers that follow a synchronous pattern, where every name resolution query is awaited, as well as with type checkers that use dynamic scheduling techniques, such as worklists and continuations.

We claim the following contributions:

- We propose the notion of *scope state* to explicitly track the presence of weakly-critical edges (Section 5.3.3).
- We introduce a model of hierarchical compilation units with scope sharing (Section 5.4.1). We extend scope state with a notion of sharing, which allows us to track weakly-critical edges in the hierarchy of compilation units (Section 5.4.2).
- We present a scope graph-based name resolution API for use by type checker implementations (Section 5.4.3).
- We present an actor-based algorithm that implements the hierarchical compilation unit model and the name resolution API, and provides implicit parallel execution of the compilation units (Section 5.5).
- We present a fine-grained deadlock handling approach to ensure termination that is well-suited for interactive applications of the type checkers (Section 5.5).
- We show that our framework captures the scheduling behavior of Rouvoet, van Antwerpen, et al. (2020a) by porting the Statix solver to our framework. We discuss local inference and the need for a specification style that models declarations and their types as separate scopes (Section 5.6). We parallelize all Statix type checkers, provided they follow this specification style.
- We benchmark the parallel Statix solver using a specification for a subset of Java on a few real world projects, showing speedups up to 5.0x on 8 cores for larger projects.

All the source code and benchmark results are available in the accompanying artifact.

```
package p;

class A {
    p.C f;
    static class C {}
}
```

```
package p;

class B extends A {
    C g;
}
```

```
package p;

class C {
    A h;
}
```

Figure 5.1: Three unit Java program demonstrating mutual and discovered dependencies.

5.2 Motivation and Scope

Our goal is to develop a framework that provides implicit parallelization of type checkers. In this section we discuss the features we want to support, the difficulties these features pose to parallelization, and an overview of our solution.

We use an example of a Java program consisting of three compilation units, shown in Figure 5.1, to illustrate the requirements on parallel type checkers. This example shows to two dependency patterns that are challenging for parallelization. The first is *mutual dependencies*. Class `A` refers to class `p.C` (qualified to distinguish it from the nested class it defines), while class `C` refers to class `A`. The second is *dynamic dependencies*. These are dependencies that are discovered during type checking, and that are not obvious without at least partially checking the program. The reference from `B` to `C` is an example of this. The name `C` could refer to the top-level class in the package, or to the nested class in `A`. To decide that it refers to the nested class in `A`, we need to resolve the reference to the super class `A` and its interface.

Typical compiler design (e.g., Appel, 1998) divides compilation into phases, including parsing, type checking, and code generation. We focus on the type checking phase, which is often difficult to parallelize because of the context dependence of type checking and name resolution. The type checking phase of the compiler for our example may consist of several steps: (a) build a symbol table containing information on defined classes and type inheritance, (b) build the interface for each class by processing field and method declarations, and (c) check the field and method bodies of each class. Each step depends on the information collected in the previous phases.

What does it take to parallelize this type checker? Compilation units are checked in parallel, but inevitably need information provided by other compilation units. Mutual dependencies between compilation units prevents linear ordering of compilation units, while dynamic dependencies mean part of the work must be done before all dependencies are even known. This immediately rules out simple parallelization schemes based on a topological ordering of compilation units, where units are checked after their dependencies have finished.

The main challenge introduced by parallelization is therefore how to deal with partial information during type checking (which has been called the Doesn't Know

Yet Problem; Seshadri et al., 1988). For example, the compilation unit for class `B` does a lookup of nested classes in its super class `A`, while the compilation unit for `A` has not constructed its interface yet. Solving this problem may require designing locking schemes for threads with shared data, or messaging protocols for communicating processes, as well as keeping track of the completeness of (part of) the symbol table or interface. Designing concurrent software is notoriously hard, and bugs can result in deadlocks or invalid results because of the use of incomplete information.

Our solution to this problem is a framework that allows compiler engineers to write their type checker without concern for parallelization. All the work of coordinating the parallel units and keeping track of completeness of interfaces is handled by the framework. The key idea is that all dependencies between units are the result of either a hierarchy between units (e.g., compilation units in packages) or name[2] lookups. The framework provides the type checker with a name handling API based on the expressive name binding model of scope graphs. Scope graphs are language-independent, and have been successfully used to model a wide variety of binding patterns, including mutually recursive modules, type-dependent names, and generics. Type checkers use this API to declare the name binding and scoping structure, and resolve names by queries on the resulting scope graph. The scope graphs grows monotonically with the type checker marking the parts of the graph that are completed. In return, the framework completely hides the communication between different compilation units and ensures only complete information is used to answer name resolution queries.

In terms of our earlier example, this means that the type checker of each compilation unit follows the steps of the original, non-parallel, design. When the type checker for class `B` queries the not-yet-constructed interface of class `A`, the query is simply suspended until the unit of `A` has progressed enough to be able to answer the query. The type checker of unit of `A`, on the other hand, is unaware of the query as the delay and answer mechanism is handled transparently based on the monotone completion of `A`'s scope graph.

5.3 Type Checking with Scope Graphs

Scope graphs (Néron et al., 2015) are a language-independent formalism to specify name binding and name resolution, and a key ingredient of our approach. In this section we explain scope graphs, describe the problem of safe name resolution and its solution using critical edges (Rouvoet, van Antwerpen, et al., 2020a), and we introduce the notion of scope state to track the presence of critical edges.

[2] We use name in a broad sense, as it can be complex data and does not necessarily have to appear literally in the original program.

5.3.1 Scope Graphs

Scope graphs describe the name binding structure of programs as a directed graph of scopes with associated data, connected by labeled edges. Scopes correspond to regions in the program that behave uniformly with respect to name binding. Name resolution corresponds to queries in the graph that find paths from references to scopes with matching associated data (e.g., an identifier name).

The program in Figure 5.2 will be our running example. It consists of two files, one containing a class `A` in a package `p`, and a class `B` in a package `q`. Class `B` extends class `A` and refers to a field `f` in `A` from the method `m`. The scope graph corresponding to our example is shown in Figure 5.3.[3] The node labeled s_R represents the root scope of the program. The scopes s_p, s_q, s_A, s_B, s_f, and s_m correspond to declarations in the program, and each has the simple class name as associated data, depicted as $s \to x$. Edges from the containing scopes to these declarations are labeled to indicate the kind of declaration: `PKG`, `CLS`, `FLD` and `MTHD` for package, class, field, and method declarations respectively. The scopes $s_{T(f)}$ and $s_T(m)$ represent the types of the declarations `f` and `m`, and each is connected to their declaration with a `TYPE`-labeled edge. The label `LEX` is used for connecting a scope to its lexical parent.[4] Finally, class extension is modeled by an `EXT`-labeled edge between the two class scopes, which makes the declarations from the super-class reachable from the subclass.

Name resolution is expressed by means of queries over the scope graph, finding a path from a reference's scope to a matching declaration. Query parameters control *reachability* and *disambiguation*. What data in the graph is reachable is specified with a regular expression describing valid paths and a predicate describing matching data. For example, to resolve a reference `f` in the lexical context or in a superclass, one would use the regular expression $\texttt{LEX}^*\texttt{EXT}^*\texttt{FLD}$, and a predicate matching the name `f`. Disambiguation determines which data is visible if multiple reachable results are found, and is specified with an order on edge labels and a predicate describing equivalent data. For example, if we prefer local definitions of field references over definitions that traverse more edges, and definitions found in super-classes over definitions found in the lexical context, we would use a label order $\$ < \texttt{EXT} < \texttt{LEX}$. The special label \$, assumed different from all user-provided labels, is used for end-of-path.

We use the following notation: A scope graph $\mathcal{G}$ is a triple $\langle S, E, \rho \rangle$ of scope identifiers S, edges E, and a partial function ρ from scopes to associated data. The function

[3] This scope graph is a simplification of the scope graph that would be necessary to support all Java's name resolution features, and does not account for the possibility of named and wildcard imports, implicit package visibility, nested classes, etc.

[4] We say binding is *lexical* if binders are only visible in sub-terms of the term where they are bound. Examples are lambda and let expressions. If the scope of the binder is wider, we say the binding is *non-lexical*. Examples are identifiers imported from modules, or references to members on expressions of a class/record type.

```
package p;
public class A {
    int f;
}
```

```
package q;
public class B extends p.A {
    public int m() {
        return f;
    }
}
```

Figure 5.2: Example Java program with two compilation units.

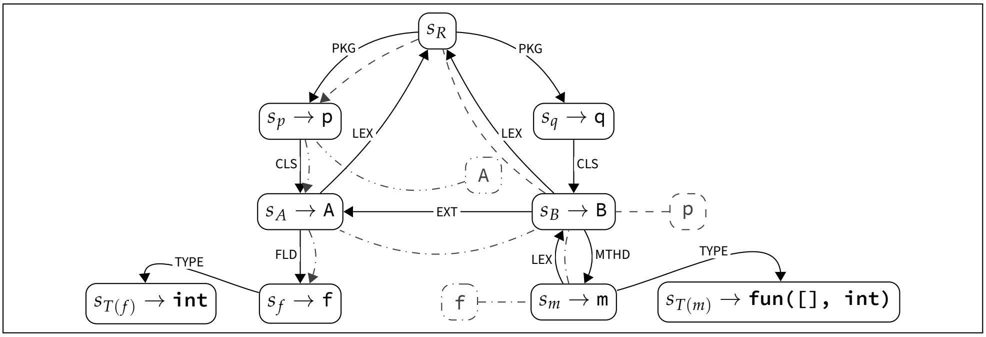


Figure 5.3: Scope graph corresponding to the Java program in Figure 5.2.

scopes($\mathcal{G}$), *edges*($\mathcal{G}$), and *data*($\mathcal{G}$) project the three components of the triple. Query parameters are a path well-formedness regular expression *re*, a data matching predicate $\mathbf{D}$, and a strict partial order on labels $\mathbf{L}$. We use $\mathbf{D}_{\mathsf{x}}$ for the predicate matching the name x. The result of a query is an answer set A of tuples (p, d) of a path p and a datum term d. A path p is either a single scope s, or a labeled step $p \cdot l \cdot s$, and *target*(p) projects the last scope of path.

5.3.2 Critical Edges for Safe Name Resolution

Type checkers use the scope graph to resolve names, but must also construct the scope graph. When type checking starts, all we have is an empty scope graph, which is gradually built up as type checking progresses. Rouvoet, van Antwerpen, et al. (2020a) observe that it is not always possible to construct the full scope graph without already querying it. This can be seen in our example in Figure 5.3 as well. The construction of the extension edge from s_B to s_A requires resolving the reference to A in a partial graph. This raises the question when it is safe to do so. After all, if that query was executed before the declaration of A is added to the graph, it results in an undeserved error. A query is considered *safe* to resolve when its result in the current partial scope graph is the same as its answer in the final scope graph. Rouvoet, van Antwerpen, et al. (2020a) identify the absence of *critical edges* as the condition to guarantee safety. A critical edge is the first edge that is missing in the partial graph that will be part of the query result in the final graph. For example, if the graph in Figure 5.3 was complete except for the `EXT` edge between s_B and s_A, this edge is critical for the resolution of the reference `f`. But the same missing edge is not critical for the resolution of `p`.

Since determining the critical edges in an incomplete scope graph amounts to solving the whole name resolution problem, Rouvoet, van Antwerpen, et al. (2020a) propose *weakly critical edges* as a conservative approximation of critical edges. Weakly critical edges are missing edges that *may* lead to a result for the query. In our earlier example, the missing `EXT` edge is weakly critical for the resolution of `f` even if s_A does not eventually contain a declaration for `f`.

5.3.3 Scope States

The final step to guarantee safe resolution is then to determine the weakly critical edges. The solution of Rouvoet, van Antwerpen, et al. (2020a) is a predicate over a constraint set, which is specific to the Statix meta-language. The crucial property of their safety predicate is that the set of weakly critical edges only decreases as type checking progresses. They prove that this ensures that once a query is executed, there can not be additions to the scope graph that lead to new results for that query. Our purpose is to develop a language-independent framework for parallel type checkers that correctly handles the dependencies between compilation units. Dependencies are the result of name resolution, thus handling the dependencies between units means ensuring name resolution between units is correct. If we can capture the presence of weakly critical edges independently of the particular type checker and object language, we can provide a general mechanism to delay queries until they are safe to execute.

To that purpose, we introduce the notion of *scope state*, which consists of a *state* (open, closing, and closed) and a set of *open labels* O, consisting of edge labels l or the special data label \$. Then, weakly critical edges are characterized by scopes that are open and/or have open edges. When a scope is closed, it is always safe to query, since its associated data and outgoing edges are final. When the scope is closing, queries over labels that are not weakly critical, i.e. that are not in O, are still safe to execute. The idea is that once the scope is closing, the set of open labels only decreases, and only labels in O are weakly critical.

Figure 5.4 shows the state transition diagram for scope states, and Figure 5.5 shows the pre- and post-conditions for the transitions. Initially, scopes are in the open state. In this state, the set of open labels has not been initialized, so all labels are considered weakly critical. Initialization with $\mathit{initScope}(d, L)$ changes the state to closing. The flag d specifies if the scope will have associated data. The set of edge labels L determines which labels outgoing edges from this scope may have. In the state closing, the set of open labels, and therefore the set of weakly critical edges, only

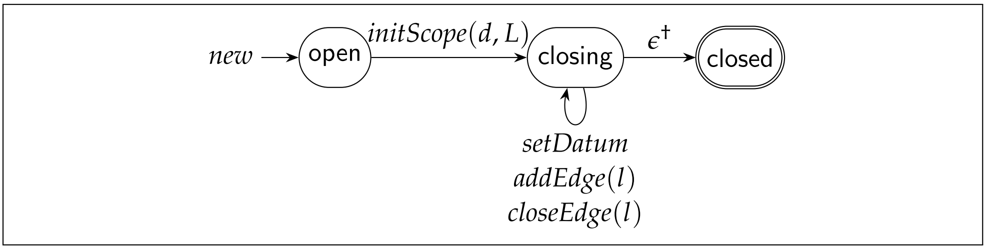


Figure 5.4: Scope states. Transition diagram.

$$\begin{array}{rcl}
\{\top\} & \mathit{new} & \{O = \emptyset\} \\
\{\top\} & \mathit{initScope}(d, L) & \{O = L \uplus \{\$ \mid d = \top\}\} \\
\{\$ \in O\} & \mathit{setDatum} & \{\$ \notin O\} \\
\{l \in O\} & \mathit{addEdge}(l) & \{\top\} \\
\{l \in O\} & \mathit{closeEdge}(l) & \{l \notin O\} \\
\{O = \emptyset\} & \epsilon^{\dagger} & \{\top\}
\end{array}$$

Figure 5.5: Scope states. Transition conditions and effects.

decreases. The preconditions on *setDatum* and *addEdge*(l) guarantee that the shape of the scope with respect to a label l only changes when the label is in the set of open labels O. The associated data of a scope can only be set once, therefore *setDatum* always removes the data label \$ from the set of open labels. Edge labels are closed with *closeEdge*(l), which removes that label from the set of open labels, after which no new edges with that label are allowed, and the label is not weakly critical anymore. After all labels have been closed, the scope is complete and in the state closed.

5.4 Hierarchical Compilation Units

In this section we introduce a model of hierarchical compilation units. We extend scope states with a notion of sharing that is required by this model. Finally, we present the API of our framework, and code samples for the type checkers that may check our running example.

5.4.1 The Compilation Unit Model

We propose a model of hierarchical compilation units. The goal of this model is to be flexible enough to handle many different project structures. Examples of typical project structures that we support are:

- a flat set of compilation units for the source files in the project, each of which introduces global declarations that are accessible from other source files,

- a tree of compilation units, where intermediate nodes represent packages or modules, and the individual source files are the leaves, or
- a project which depends on a library that is otherwise independent of the project.

Each compilation unit in our model has an associated, user defined, type checker. Compilation units can spawn sub-units, with their own associated type checkers. Each compilation unit builds a local scope graph by creating scopes, setting data, and adding edges. The compilation units are connected via scopes that are shared between units and their sub-units. Shared scopes allow sub-units to provide declarations that are reachable for other units, and to resolve to names in other units. The examples above all fit into this model. A project with a flat structure consists of one project unit, which creates a global scope that is shared with all file units, which add globally reachable declarations to the global scope. A project with hierarchical packages has compilation units for each package level, each with their own package scope, which is declared in the scope of the parent package. In a project consisting of a library and a program that depends on it, the program and the library have their own root scopes, and the dependency is reflected by an edge from the program root scope to the library root scope.

The compilation units for our Java example of Figure 5.2 follow the package hierarchy. Figure 5.6 shows how the scope graph of our example is distributed over compilation units. Our example program has five compilation units, which are depicted by the dashed boxes. At the top level, surrounding the whole scope graph, is the unit that represents the whole program. The packages p and q are sub units, each containing the units for the class in that package. The *owner* of a scope or edge in the graph is the unit that created the scope or edge. For example, the root scope s_R is owned by the root unit, while the class scope s_B is owned by the innermost unit for the class B. Visually, a scope is owned by the innermost unit that contains it, while an edge is owned by the innermost unit that contains the edge label. The MTHD edge is therefore owned by the unit of B, as is the LEX edge to s_R.

5.4.2 Safe Name Resolution with Sharing

The connection between units and sub-units is established through scopes that are shared from a unit to its sub-units. As a result, multiple units may contribute to a scope, something that the unit owning the scope must take into account when handling queries in that scope. Therefore, we extend scope state with sharing, to account for the fact that scope state is determined by the owner as well as by sub-units with which the scope was shared.

Sharing can result in outgoing edges having a different owner than their source scope, as units can contribute outgoing edges to either their own scopes, or scopes

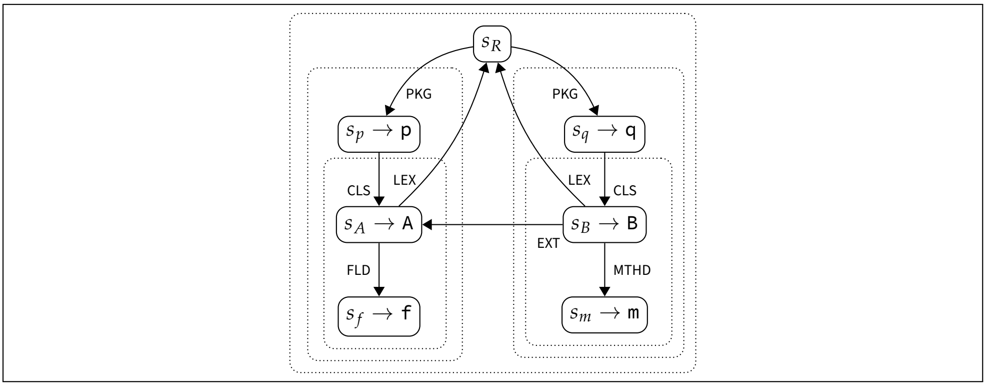


Figure 5.6: Compilation units for the Java program of Figure 5.2. The dashed boxes indicate the boundaries of the compilation units. A scope is owned by the innermost unit in which it appears. Edges are owned by the innermost unit that contains their label.

owned by one of their enclosing units that are *shared* with the unit. In our example, the CLS edge to s_B has source scope s_q, which is owned by the unit of package q, not by the unit of B. The data associated with scopes can only be provided by the owner.

The compilation unit that owns a scope is responsible for executing queries on that scope. Every unit maintains an aggregate view of each scope it owns, consisting of all the edges contributed by itself or by sub-units that the scope is shared with. To ensure safe name resolution in this model, we must account for sharing of scopes between units. When a unit initializes one of its scopes, it does not necessarily know what edges any of the sub-units may contribute. The sub-units must therefore initialize shared scopes as well, so that the scope owner has a complete picture of the state of the scope. In the previous section, a scope moved immediately to the closing state when it was initialized. When the scope is, or can be, shared, this is not correct. When a scope is not in state open, we expect that the set of open labels only decreases. The initialization of the scope by a sub-unit could increase the set of open labels. On top of that, if the scope is shared with a new sub-unit, this sub-unit must now also initialize the scope, potentially adding open labels. We can be sure that the set of open labels will only decrease, when all units that the scope is shared with have initialized it, and none of those units will share the scope with new sub-units.

To handle sharing correctly, we extend scope states with an explicit notion of sharing. The state diagram for scope states with sharing is shown in Figure 5.7, and the pre- and postconditions for the transitions in Figure 5.8. The extended scope state consists of a set of open labels per unit O, a set I of units that must initialize the scope, and a set H of units that may share the scope with new sub-units. All transitions take a parameter u that indicates which unit is responsible for the event. Creation of a scope is indicated by $new(u, d)$, where u is the owner and the flag d indicates

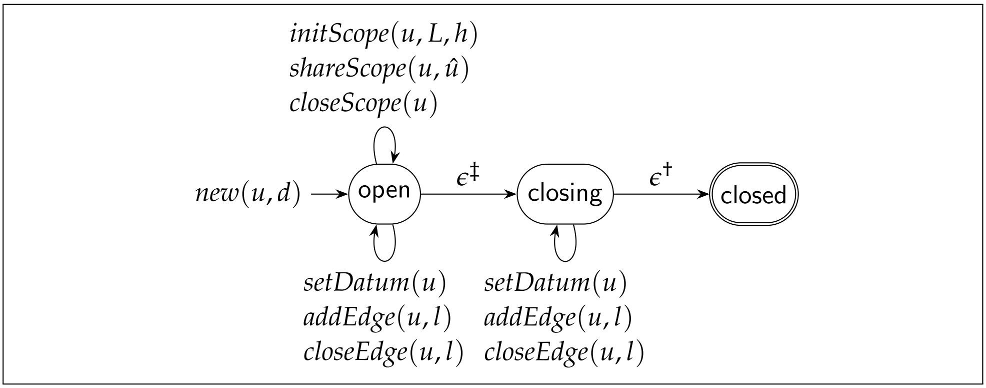


Figure 5.7: Scope states with sharing. Transition diagram.

$\{\top\}$	$new(u, d)$	$\left\{\begin{array}{l} I = \{u\}, \\ O = \{\$ \mid d = \top\}, \\ H = \emptyset \end{array}\right\}$
$\left\{\begin{array}{r} u \in I, \\ O = O', \\ u \notin H \end{array}\right\}$	$initScope(u, L, h)$	$\left\{\begin{array}{l} u \notin I, \\ O = O' \uplus \{(u, l) \mid l \in L\}, \\ u \in H \end{array}\right\}$
$\left\{\begin{array}{r} u \in H, \\ \hat{u} \notin I \end{array}\right\}$	$shareScope(u, \hat{u})$	$\{\hat{u} \in I\}$
$\{u \in H\}$	$closeScope(u)$	$\{u \notin H\}$
$\{(u, \$) \in O\}$	$setDatum(u)$	$\{(u, \$) \notin O\}$
$\{(u, l) \in O\}$	$addEdge(u, l)$	$\{\top\}$
$\{(u, l) \in O\}$	$closeEdge(u, l)$	$\{(u, l) \notin O\}$
$\{I = \emptyset, H = \emptyset\}$	$\epsilon^{\ddagger}$	$\{\top\}$
$\{O = \emptyset\}$	$\epsilon^{\dagger}$	$\{\top\}$

Figure 5.8: Scope states with sharing. Transition conditions and effects.

```
interface TypeChecker
    function run(S)
end
interface CompilationUnit
    function freshScope(d) : s
    function addSubUnit(tc, S)
    function initScope(s, L, h)
    function closeScope(s)
    function setDatum(s, d)
    function addEdge(s, l, s')
    function closeEdge(s, l)
    async function query(s, re, D, L) : A
end
```

Figure 5.9: Type Checker and Name Resolution API.

whether the scope has associated data. The flag d is not part of *initScope* anymore, because only the owner can set data, but the scope is initialized by all units that the scope is shared with. When a scope is shared with a unit $\hat{u}$ with $shareScope(u, \hat{u})$, that unit is added to the set I. Every unit that the scope is shared with must initialize it with $initScope(u, L, h)$, after which it is removed from I. A unit initializes the scope with the set of open labels L, a well as the flag h to indicate that the unit may share the scope with sub-units. The scope moves to the state closing when the set of uninitialized units I and the set of sharing units H are both empty. When the state is not open anymore, the set of open labels will only decrease. The events $setDatum(u)$ indicates associated data is set, while $addEdge(u, s, l, s')$, and $closeEdge(u, s, l)$ indicate adding an edge and closing an edge label. The preconditions require that these events are only coming from units that have already initialized the scope. Note that these events are allowed in the states open *and* closing. This ensures that, if a scope is shared between multiple units, each unit can extend that scope without having to wait for all other units to initialize the scope first.

5.4.3 Name Resolution API

The key to support parallel execution of type checkers is to correctly handle the dependencies between compilation units, which result from name resolution. Queries into a unit that has not constructed the relevant part of its scope graph must be delayed, and executed whenever the scope graph is complete enough. Our framework hides this scheduling from type checkers, and thus provides implicit parallel execution. Type checkers are programmed against a name resolution API, shown in Figure 5.9, which contains methods to specify name binding by building a unit's scope graph, resolve names by querying the scope graph, and start sub-units.

Names are resolved with the $query(s, re, \mathbf{D}, \mathbf{L})$ function, which is defined as *async* to reflect the fact that queries cannot always be answered directly by other compilation units. It is up to the type checker to decide if the result should be immediately awaited, or if other work can be done until the answer is available. The framework ensures correct query answers by keeping track of scope states and scheduling queries based on these scope states. Type checkers are responsible for providing the framework with the necessary information to maintain the scope state. The type checker must therefore initialize the set of (locally) open labels and announce whether it may share the scope with sub-units, and it must close edge labels once all edges with that label are added. All the interaction with other units, such as forwarding queries to the right unit, delaying queries, and maintain scope state on sharing is completely hidden from the type checker. For example, the function $addSubUnit(tc, S)$, which starts a sub-unit with the given type checker tc and initial scopes S, takes care of recording the sharing of scopes, and starts the type checker to run in parallel. Type checkers specify what they do locally, the framework implicitly takes care of their parallel execution.

The pseudo code in Figure 5.10 shows how the API could be used to implement a type checker that checks the Java running example.[5] Each type checker is an actor that extends the *CompilationUnit* actor that provides the API, which is explained in detail in Section 5.5. At the top level is *JavaRootTC*, which takes no scope arguments, and creates the root scope of the project. Initialization specifies no open edge labels, but does allow sharing. For each package a new sub unit is started with a package type checker that takes the root scope as argument. The first is the root scope, which is passed down to the class scopes. After creating the sub units, the scope is closed, to indicate it will not be shared anymore. The package type checkers start by initializing the shared root scope, indicating the scope may be shared with sub units, and marking `PKG` as open to allow adding the package declaration. A new package scope s_p is created, with the package name as associated data. The root scope and package scope are shared with the sub units for the classes in the package, after which both scopes are closed. Finally, the package declaration is added to the root scope and the `PKG` label is closed.

Although not immediately evident in this small example, the fact that the API is fine-grained (e.g., separating closing a scope for sharing from closing an open edge label) allows greater flexibility in how the type checker is implemented than when a type checker would be responsible for aggregating all these events until one final event can be constructed.

[5] We show all API calls directly, to show how the API can be used. We imagine that in an actual type checker implementation, common patterns of usage would be abstracted away for nicer code.

```
actor JavaRootTC(P) extends TypeChecker
    function Run({})
        $s_R := freshScope(\bot)$
        $initScope(s_R, \emptyset, \top)$
        for each $(p, C) \in P$ do
            $addSubUnit(JavaPkgTC(p, C), \{s_R\})$
        $closeScope(s_R)$
    end
end
actor JavaPkgTC(⟦package x;⟧, C) extends TypeChecker
    function Run({$s_R$})
        $initScope(s_R, \{\text{PKG}\}, \top)$

        $s_p := freshScope(\top)$
        $initScope(s_p, \emptyset, \top)$
        $setDatum(s_p, x)$
        for each $c \in C$ do
            $addSubUnit(JavaClassTC(c), \{s_R, s_p\})$
        $closeScope(s_R)$
        $closeScope(s_p)$

        $addEdge(s_R, \text{PKG}, s_p)$
        $closeEdge(s_R, \text{PKG})$
    end
end
actor JavaClassTC(⟦class x extends y { ... }⟧) extends TypeChecker
    function Run({$s_R, s_p$})
        $initScope(s_R, \emptyset, \bot)$
        $initScope(s_p, \{\text{CLS}\}, \bot)$

        $s_c := freshScope(\top)$
        $initScope(s_c, \{\text{LEX}, \text{EXT}, \text{FLD}, \text{MTHD}\}, \bot)$
        $setDatum(s_c, x)$
        $addEdge(s_c, \text{LEX}, s_R)$
        $closeEdge(s_c, \text{LEX})$

        $addEdge(s_p, \text{CLS}, s_c)$
        $closeEdge(s_p, \text{CLS})$

        $\{(p, z)\} :=$ await $query(s_c, \text{LEX}^*\text{CLS}, \mathbf{D}_x, \ldots)$
        $s'_c := target(p)$
        $addEdge(s_c, \text{EXT}, s'_c)$
        $closeEdge(s_c, \text{EXT})$

        // ... etc ...
    end
end
```

Figure 5.10: Sketch of a simplified type checker implementation for Java packages and classes. The type checker is defined as compilation units *JavaRootTC* for the project root, *JavaPkgTC* for packages, and *JavaClassTC* for classes. The presented code shows the construction and querying of package and class definitions.

$$\begin{array}{rcl}
\mathit{msg} & := & \mathit{Start}(S) \mid \mathit{InitScope}(s, L, h) \mid \mathit{ShareScope}(s) \mid \mathit{CloseScope}(s) \\
 & \mid & \mathit{AddEdge}(s, l, s') \mid \mathit{CloseLabel}(s, l) \mid \mathit{Query}(s, re, \mathbf{D}, \mathbf{L}) \\
 & \mid & \mathit{DeadlockQuery}(u, m) \mid \mathit{DeadlockReply}(u, m, U) \mid \mathit{Deadlocked}(U) \\
\mathit{token} & := & \mathsf{initScope}(s) \mid \mathsf{closeScope}(s) \mid \mathsf{closeLabel}(s, l) \mid \mathsf{answer}(f)
\end{array}$$

Figure 5.11: Compilation Unit. Messages and wait-for tokens.

The pseudo code for *JavaClassTC* shows a pattern in which scope graph construction and querying are interleaved. The query for the super class is executed and the type checker waits for the result to be able to construct the EXT edge between the class scopes. It is important to realize that, because the framework ensures safe name resolution, this also introduces the possibility of *deadlock*. If, for example, the *JavaClassTC* type checker would postpone $\mathit{closeEdge}(s_c, \mathsf{LEX})$ until after awaiting the query result, the query would get stuck on the still open edge label. It is therefore important to realize that type checker developers are still responsible for scheduling concerns that are part of any compiler implementation (concurrent or not), such as ensuring declarations are introduced before they are queried. The framework cannot solve these issues, as they are dependent on the specifics of the object language, but it ensures the local behavior is preserved when run in parallel. The Statix meta-language (Rouvoet, van Antwerpen, et al., 2020a) provides implicit maintenance of scope state and flexible scheduling as part of the meta-language semantics, so that these concerns can be left implicit in a Statix type system specification. The case study in Section 5.6 shows that it is possible to implement a Statix solver on top of our framework, which gives the best of both worlds: implicit parallelism *and* implicit handling of scope state and scheduling.

5.5 Parallel Actor-Based Algorithm

In this section we present an algorithm that implements the compilation unit model and API that were introduced in the previous section. First we introduce the actor model that forms the basis of our algorithm, then we discuss the three main aspects of the algorithm:

- maintaining the scope graph and scope states for owned and shared scopes,
- safely resolve queries on own scopes and delegate queries on other scopes, and
- detect deadlock between compilation units to ensure termination.

5.5.1 Compilation Unit Actors

The algorithm is written following the actor paradigm (Agha, 1990). Actors are a concurrency model based on message passing. An actor has only local state, and

```
actor CompilationUnit()
    var: scope graph $\mathcal{G}$
    var: counting wait-for graph WFG
    var: delays $\mathcal{Z} := \emptyset$
    abstract function run(S)
```

Figure 5.12: Compilation Unit. Local actor state.

communicates with other actors through messages. Actors are not internally concurrent, and they do not share state. This makes reasoning about concurrency easier with actors than with approaches based on shared state and explicit synchronization.

A compilation unit corresponds to a *CompilationUnit* actor, which definition and local state is shown in Figure 5.12. The members of a *CompilationUnit*, which are introduced in the following sections, are shown in Figures 5.13, 5.15, 5.16 and 5.20. The local state of compilation units consists of a scope graph $\mathcal{G}$, a counting wait-for graph *WFG*, and a set of delayed queries $\mathcal{Z}$. The messages that form the protocol between compilation units are listed in Figure 5.11. Type checkers are implemented by extending the actor and implementing the abstract *run* method.

Since there are many variations of the actor model, we give a quick overview of the features that we assume in the model:

- Actors are started using **start**, and form a hierarchy. The keywords **self** and **parent** refer to the current actor or its parent actor, respectively. Actor references can be sent to other actors.

- Actors implement **receive** members for all messages that they accept. Inside a message handler, the **sender** keyword refers to the sender of the current message. Messages are sent using **send** $actor, msg$. Messages that require a response are sent with **request** $actor, msg$ and the response is sent from the message handler with **reply** msg. Messages from one actor to another are delivered in order, but delivery of messages from different actors is arbitrarily interleaved.

- Actors may implement auxiliary **function** members, which can only be invoked locally.

Some algorithms are presented in an asynchronous style, using futures. They use the following primitives:

- A **future** f represents a value that may be provided later. The value of a future is set by applying it, written as $f(v)$.

- Functions can be marked as **async** to indicate that they return a future. Inside asynchronous functions, the **await** keyword is used to await the results of futures.
- Awaited futures do not block the actor, but suspend the currently handled message and allow other messages to be processed by the same actor. A resumed computation (as a result of a reply or an applied future) always runs in the context of the actor that started it, and never concurrently with message handling or other resumed computations.

The message handlers and functions of the type checker API are implemented in a straight-forward way. The handler for the message $\mathit{AddEdge}(s, l, s')$ calls $\mathit{addEdge}(\mathbf{sender}, s, l, s')$, and the API function $\mathit{addEdge}(s, l, s')$ calls $\mathit{addEdge}(\mathbf{self}, s, l, s')$.

5.5.2 Maintaining Scope Graph and Scope States

A compilation unit locally maintains its scope graph $\mathcal{G}$ and the states of the scopes it owns. This is done by the group of functions shown in Figure 5.13. These functions are called to handle API calls from the local type checker, or to handle messages received from other units. In the former case, the argument u equals **self**, in the latter u equals **sender**.

Scope state is maintained in a wait-for graph *WFG*, which consists of edges between units, labeled by a token indicating an expected action from the target unit. The tokens that may appear in the wait-for graph are listed in Figure 5.11. The state of the sets I, H, and O of the scope state is determined by the tokens in the wait-for graph. An $\mathsf{initScope}(s)$ edge to u implies $u \in I_s$. A $\mathsf{closeScope}(s)$ edge to u implies $u \in H_s$. A $\mathsf{closeLabel}(s, l)$ edge to u implies $(u, l) \in O_s$. The state of a scope s can be determined from the tokens in the wait-for graph. If the graph contains $\mathsf{initScope}(s)$ or $\mathsf{closeScope}(s)$ tokens, the scope is open. If the graph only contains $\mathsf{closeLabel}(s, l)$ tokens, the scope is $\mathsf{closing}$. If there are no tokens concerning s, the scope is closed.

The functions in Figure 5.13 update the wait-for graph in correspondence with the postconditions of the scope state transitions. When an element is added to one of the sets of the scope state, an edge is added with $\mathit{waitFor}(u, \mathit{token})$. When an element is removed from one of the sets of the scope state, an edge is removed with $\mathit{granted}(u, \mathit{token})$. For example, when a fresh scope is created with *freshScope*, the function adds an $\mathsf{initScope}(s)$ token, corresponding to the postcondition $u \in I$ of *new*. When the scope is initialized with $\mathit{initScope}(s, L, h)$, the $\mathsf{initScope}(s)$ is removed, corresponding to the postcondition $u \notin I$ of *initScope*.

The removal of tokens from the wait-for graph may result in changes to the weakly critical edges of a scope. Therefore, the functions *initScope*, *closeScope*, and *closeLabel* call one of the *tryRelease∗* functions to trigger the release of queries that can now be executed safely.

```
function start(S)
    $\mathcal{G} := \mathcal{G} \uplus \langle S, \emptyset, \emptyset \rangle$
    foreach $s \in S$ do waitFor(self, initScope($s$))
    run($S$)
end
function freshScope(u, d)
    pick $s$ fresh in scopes($\mathcal{G}$)
    $\mathcal{G} := \mathcal{G} \uplus \langle \{s\}, \emptyset, \emptyset \rangle$
    waitFor($u$, initScope($s$))
    if $d = \top$ then waitFor($u$, closeLabel($s$, \$))
    return $s$
end
function initScope(u, s, L, h)
    granted($u$, initScope($s$))
    foreach $l \in L$ do waitFor($u$, closeLabel($s, l$))
    foreach $i \in 0 \ldots h$ do waitFor($u$, closeScope($s$))
    if owner($s$) = self then tryReleaseScopeDelays($s$)
    else send parent, InitScope($s, L, h$)
end
function addSubUnit(u, û, S)
    foreach $s \in S$ do shareScope($\hat{u}$, $s$)
    start $\hat{u}$
    send $\hat{u}$, Start($S$)
end
function shareScope(u, s)
    waitFor($u$, initScope($s$))
    if owner($s$) $\neq$ self then send parent, ShareScope($s$)
end
function closeScope(u, s)
    granted($u$, closeScope($s$))
    if owner($s$) = self then tryReleaseScopeDelays($s$)
    else send parent, CloseScope($s$)
end
function setDatum(u, s, d)
    $\mathcal{G} := \mathcal{G} \uplus \langle \emptyset, \emptyset, \{(s, d)\} \rangle$
    closeLabel($u$, $s$, \$)
end
function addEdge(u, s, l, s')
    $\mathcal{G} := \mathcal{G} \uplus \langle \emptyset, (s, l, s'), \emptyset \rangle$
    if owner($s$) $\neq$ self then send parent, AddEdge($s, l, s'$)
end
function closeLabel(u, s, l)
    granted($u$, closeLabel($s, l$))
    if owner($s$) = self then tryReleaseLabelDelays($s$, $l$)
    else if $l \neq \$$ then send parent, CloseLabel($s, l$)
end
```

Figure 5.13: Compilation Unit. Scope graph.

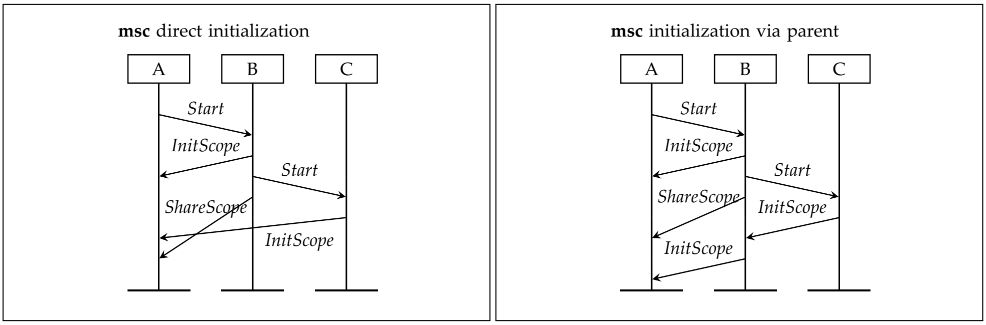


Figure 5.14: Different initialization scenarios.

Maintaining the scope graph and state locally is not enough for a shared scope s that is not owned by the current unit. In such cases, when $\mathsf{owner}(s) \neq \mathbf{self}$, the event is propagated to the parent. Because scopes can only be shared with sub units, this means that the message eventually reaches the owner of that scope. The benefit of propagating the message via the parent instead of sending it to the owner of the scope directly has to do with message ordering. Messages coming from two different units are not meaningfully ordered. This can lead to messages arriving in unexpected order, as illustrated by the two scenarios in Figure 5.14. Without message ordering, a scenario where a top-level unit A, shares a scope with a sub-unit B, which in turn shares that scope with a sub-unit C, could result in A receiving the initialization of C before the message from B that the scope was shared. If the initialization goes via the parent B, then unit A always gets the *ShareScope* message before the corresponding initialization.

Receiving the messages in order simplifies maintenance of the wait-for graph and makes it easier to enforce correct usage of the API in the implementation. This is a simple solution to achieve that without the need for more complex message ordering mechanisms such as vector clocks. Sending messages about shared scopes via the parent is also the reason that the wait-for graph is a counting graph, that is, tokens may appear multiple times in the graph. To the unit A it looks as if the unit B has to initialize the shared scope twice, as it does not know about the unit C. All messages about sharing and initialization appear to come from B.

5.5.3 Resolving Queries

The name resolution algorithm, shown in Figure 5.15, implements a graph search that follows well-formed paths to matching declarations. It is a reformulation of the algorithm presented by van Antwerpen et al. (2016). The entry point is the function $query(p, re, \mathbf{D}, \mathbf{L})$, which returns the environment of paths starting with the prefix path p that matches the given query parameters. The search starts at the target scope

```
async function query(p, re, D, L)
    $u := \mathsf{owner}(target(p))$
    if $u$ = self then return await $getEnv(p, re, \mathbf{D}, \mathbf{L})$
    else
        $f :=$ request $u$, $Query(p, re, \mathbf{D}, \mathbf{L})$
        $waitFor(u, \mathsf{answer}(f))$
        $A :=$ await $f$
        $granted(u, \mathsf{answer}(f))$
        return $A$
end
async function getEnv(p, re, D, L)
    $L := \{l \mid \mathcal{L}(\partial_l re) \neq \emptyset\} \cup \{\$ \mid \epsilon \in \mathcal{L}(re)\}$
    return await $getEnvForLabels(L, p, re, \mathbf{D}, \mathbf{L})$
end
async function getEnvForLabels(L, p, re, D, L)
    $\vec{K} := \emptyset$
    $L_{max} := \{l \mid l \in L, \nexists l' \in L.\, \mathbf{L}(l, l')\}$
    for each $l \in L_{max}$ do
        $L' := \{l' \mid l' \in L.\, \mathbf{L}(l', l)\}$
        $\vec{K} := \vec{K} \cup \{getShadowedEnv(L', l, p, re, \mathbf{D}, \mathbf{L})\}$
    $\vec{A} :=$ awaitAll $\vec{K}$
    return $\bigcup_{A \in \vec{A}} A$
end
async function getShadowedEnv(L, l, p, re, D, L)
    $k_L := getEnvForLabels(L, p, re, \mathbf{D}, \mathbf{L})$
    $k_l := getEnvForLabel(l, p, re, \mathbf{D}, \mathbf{L})$
    $[A_L, A_l] :=$ awaitAll $[k_L, k_l]$
    return $shadow(A_L, A_l)$
end
async function getEnvForLabel(l, p, re, D, L)
    if $l = \$$ then
        $d :=$ await $getDatum(target(p))$
        return $\{a \mid a = (p, d), \mathbf{D}(d)\}$
    else
        $S :=$ await $getEdges(target(p), l)$
        $\vec{P} := \{p' \mid s' \in S, p' = p \cdot l \cdot s', \nexists p'.\, (p' \cdot l \cdot s \text{ prefix of } p)\}$
        $\vec{K} := \{query(p', \partial_l re, \mathbf{D}, \mathbf{L}) \mid p' \in P\}$
        $\vec{A} :=$ awaitAll $\vec{K}$
        return $\bigcup_{A \in \vec{A}} A$
end
function shadow(A_1, A_2)
    return $A_1 \cup \{(p_2, d_2) \mid (p_2, d_2) \in A_2, \nexists p_1, d_1.\, ((p_1, d_1) \in A_1, d_1 \approx d_2)\}$
end
```

Figure 5.15: Compilation Unit. Query resolution.

of the current path. If the current scope is not owned by the current unit, the query is forwarded to the owner's compilation unit. Otherwise, the environment is computed locally by *getEnv*$(p, re, \mathbf{D}, \mathbf{L})$. That function determines the set of labels L that is relevant given the current path well-formedness regular expression. Edge labels l are relevant if the Brzozowski derivative (Brzozowski, 1964) does not result in the empty language. If the current regular expression is accepting, that is, its language contains the empty string ϵ, the current scope may be an end-point, and the data label is also relevant. The functions *getEnvForLabels* and *getShadowedEnv* together ensure that the environment implements the label order specified for disambiguation, by ensuring results from more specific labels shadow results from the least specific labels. For example, if the current set of labels $L = \{\$, \mathsf{FLD}, \mathsf{LEX}, \mathsf{EXT}\}$, and the label order is $\$ < \mathsf{FLD} < \mathsf{EXT} < \mathsf{LEX}$, then the resulting environment is

$$shadow(shadow(shadow(A_{\$}, A_{\mathsf{FLD}}), A_{\mathsf{EXT}}), A_{\mathsf{LEX}})$$

A_l is the answer set for the label l, and *shadow* is the function that removes answers from the right-hand set if its datum matches any answer in the left-hand set. The environment for a single label l is computed by *getEnvForLabel*$(l, p, re, \mathbf{D}, \mathbf{L})$. If the label is the data label \$, *getDatum* is called in the current scope to construct an answer (p, d). Otherwise, *getEdges* is used to return the target scopes of all outgoing l-labeled edges, cyclic paths are filtered out to ensure search termination, and environments are resolved for each new prefix path p' with the updated path well-formedness. The result is the union of all resulting environments. The updated set of parameters is itself a valid, residual, query, which allows us to simply call the top-level *query* function, which takes care of delegating the query to the right compilation unit.

Compilation units must also ensure that name resolution is safe. When edges or data are requested for which the label is weakly critical, the answer is delayed. When a label's status changes, pending delays are released. The functions *isScopeOpen* and *isEdgeOpen* implement the check for weakly critical edges based on the wait-for graph, as explained in Section 5.5.2. The functions *getEdges* and *getDatum* decide based on the result of *isEdgeOpen* whether the scope graph can be used, or if it has to delay the answer. If the label is critical, a new future is created, which is stored, together with the scope and label, in the set of delays $\mathcal{Z}$. The functions *tryReleaseScopeDelays* and *tryReleaseLabelDelays* are called whenever the scope state changes, and return the results for any label that is not critical anymore by applying the stored future.

5.5.4 Handling Deadlock

The type checkers implemented with our framework can deadlock for various reasons. The type checker may contain obvious bugs, such as querying a scope before it is properly closed. But many subtle situations can occur as well, if ill-bound or ill-

```
function waitFor(u, token)
    $WFG := WFG \cup \{(u, token)\}$
end
function granted(u, t)
    $WFG := WFG - \{(u, token)\}$
end
function isWaitingFor(u, t)
    return $(\mathbf{self}, t, u) \in WFG$
end
function isScopeOpen(s)
    return $\exists u.\ (isWaitingFor(u, \mathsf{initScope}(s)) \vee isWaitingFor(u, \mathsf{closeScope}(s)))$
end
function isEdgeOpen(s, l)
    return $isScopeOpen(s) \vee \exists u.\ isWaitingFor(u, \mathsf{closeLabel}(s, l))$
end
async function getDatum(s)
    if isEdgeOpen(s, $) then
        future $k$
        $\mathcal{Z} := \mathcal{Z} \cup \{(s, \$, k)\}$
        return await $k$
    else return $data(\mathcal{G})(s)$
end
async function getEdges(s, l)
    if isEdgeOpen(s, l) then
        future $k$
        $\mathcal{Z} := \mathcal{Z} \cup \{(s, l, k)\}$
        return await $k$
    else return $\{e \mid e \in edges(\mathcal{G}), \exists s'.e = (s, l, s')\}$
end
function tryReleaseScopeDelays(s)
    if isScopeOpen(s) then return
    foreach $\{l \mid \exists k.\ (s, l, k) \in \mathcal{Z}\}$ do tryReleaseLabelDelays(s, l)

end
function tryReleaseLabelDelays(s, l)
    if isEdgeOpen(s, l) then return
    for each $\{k \mid ((s, l), k) \in \mathcal{Z}\}$ do
        $\mathcal{Z} := \mathcal{Z} - \{(s, l, k)\}$
        if $l = \$$ then $k(data(\mathcal{G})(s))$
        else $k(\{e \mid e \in edges(\mathcal{G}), \exists s'.\ e = (s, l, s')\})$
end
```

Figure 5.16: Compilation Unit. Delays and wait-for graph maintenance.

```
actor JavaClassTC(⟦class x extends y { ... }⟧) extends CompilationUnit
    function Run($\{s_R, s_p\}$)
        // ...
        $s_c :=$ freshScope($\top$)
        addEdge($s_p$, CLS, $s_c$)
        $A :=$ await query($s_c$, LEX*CLS, $\mathbf{D}_x, \ldots$)
        closeEdge($s_p$, CLS)
        // ...
    end
end
```

Figure 5.17: Java Type Checker with Incorrect Internal Scheduling

typed input programs cause scope graph construction to get stuck, even if no deadlocks can occur on well-typed inputs.

Whatever the reason, it is important for the user experience to ensure termination of the type checking process and the possibility of graceful handling of deadlocks by the type checker. Our goal is a fine-grained approach where deadlock is handled by failing individual queries that contribute to the deadlock, and only fail whole units as a last resort. Being fine-grained is especially important in interactive settings, when a type checker is employed as part of an IDE. Failing the type checker without returning a result because of an ill-typed input program completely negates the usefulness of the type checker to the programmer in helping them fix their program.

A deadlock occurs when a group of units waits on each other without any unit being able to make progress without receiving a message from one of the other units. We illustrate this using a faulty version of our Java type checker example, shown in Figure 5.17. In this implementation, the super class is resolved before closing the CLS label after the class declaration is added. The program causing deadlock, shown in Figure 5.18, consists of a class A and a class B that extends A, both defined in a package p. The two class definitions are checked by their own units A and B, who declare the classes in the scope s_p that is shared with them by the package unit p. Unit B tries to resolve the class A before closing the CLS edge on the shared scope s_p, and the query gets delayed on that edge by unit p. Now the units are in deadlock, since p is waiting for B to close the edge, while B is waiting for an answer from p. We can visualize the dependencies between the units by combining the wait-for graphs *WFG* of all units, as shown in Figure 5.19. We see that deadlock in the graph from the knot[6] of units that cannot make progress.

[6] In a directed graph, a knot is a set of nodes in the graph such that each node can reach all other nodes in the set. Communication deadlocks are characterized by knots, while resource deadlocks are characterized by cycles.

```
package p;
class A {}
```

```
package p;
class B extends A {}
```

Figure 5.18: Example program that deadlocks with the buggy type checker from Figure 5.17.

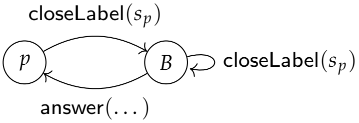


Figure 5.19: Wait-for graph for the deadlocked example in Figure 5.18.

To understand how we can handle such deadlocks in a fine-grained way, we must understand the shapes these graphs can have. The key insight is that deadlocks involving more than one unit *always* involve a query. If we do not consider queries, the structure of the wait-for graph is always a tree. Units only wait for initScope, closeScope, and closeLabel on themselves or direct sub-units. It is waiting on answers that breaks the tree structure. Therefore, a knot between different units can only exist if at least one query is involved. Our approach handles deadlocks by failing involved queries whenever possible. These failures become exceptions in the type checker, which can be handled if desired. If a deadlock does not involve any queries, and thus involves only a single unit, the whole unit is failed and any remaining open scopes and labels are closed.

The functions for deadlock handling are shown in Figure 5.20. Deadlock detection is implemented using the distributed communication deadlock detection algorithm of Chandy et al. (1983), modified so that it collects all units involved in a deadlock. When a deadlock is detected, the *deadlocked* function is called on all units involved, receiving the set U of involved units an argument. In the case that the set U is a singleton, and the deadlock is local, failing queries is attempted by *failDelays*, and, if unsuccessful, the unit is failed with *failAll*. The function *failDelays* finds all unanswered queries to units in the deadlock and raises an exception locally (indicated by the application of the future with $\bot$). The function *failAll* closes any remaining open scopes and labels and informs the parent if appropriate. At this point the type checker of the failed unit is never invoked anymore, but the unit itself can still resolve queries for other units and participate in deadlock detection. In the case that the set U is not a singleton, the *failDelays* function is used to fail any queries on other units in the deadlock. We explicitly prevent falling back to *failAll* in non-singleton deadlocks because not every unit has queries it can fail. Failing such a unit in such cases would be unnecessary, as some other units in the deadlock can fail queries and resume type checking.

```
function deadlocked(U)
    if $|U| = 1$ then
        if $failDelays(U) =$ false then failAll()
    else
        $failDelays(U)$
end
function failDelays(U)
    $Z := \{f \mid (u, \mathsf{answer}(f)) \in WFG, u \in U\}$
    foreach $Z$ do $f(\bot)$
    return $Z \neq \emptyset$
end
function failAll()
    for each $\{t \mid (u, t) \in WFG\}$ do
        granted(self, t)
        switch $t$ do
            case $\mathsf{initScope}(s)$ do
                if $\mathsf{owner}(s) \neq$ self then send parent, $InitScope(s, \emptyset,$ false$)$
                tryReleaseScopeDelays(s)
            case $\mathsf{closeScope}(s)$ do
                if $\mathsf{owner}(s) \neq$ self then send parent, $CloseScope(s)$
                tryReleaseScopeDelays(s)
            case $\mathsf{closeLabel}(s, l)$ do
                if $\mathsf{owner}(s) \neq$ self then send parent, $CloseLabel(s, l)$
                tryReleaseLabelDelays(s, l)
        end
end
```

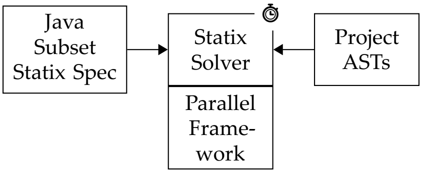
Figure 5.20: Compilation Unit. Deadlock handling.

5.6 Evaluation

We evaluated our approach by porting an existing scope graph-based type checker for a subset of Java to our framework, and measuring speedup resulting from using multiple cores when analyzing Java projects. The diagram in Figure 5.21 summarizes the setup of the benchmark. The benchmark executable, source code, and data of our experiments can be found in the artifact that accompanies this paper.

5.6.1 Benchmark

We implemented the type checker by porting the solver of the Statix meta-language to our framework. Adapting the Statix solver was an attractive case study, because it is a mature project that already uses scope graphs for name resolution. The Statix solver uses dynamic scheduling for constraint solving, where constraints are delayed on logical variable instantiation, and was thus a good test case to show that the API



Project	#Files	LOC
commons-csv 1.7	12	1845
commons-io 2.6	118	9984
commons-lang3 3.11	210	29642
single-unit-clusters-call	100	~32K

Figure 5.21: Benchmark setup

provided by the framework is flexible enough to cope with such dynamic scheduling. Therefore, we expect that type checkers in many different scheduling styles can be implemented with our framework, which we plan to explore in future work.

The type checker used a Statix specification for a subset of Java based on an existing MiniStatix specification (Rouvoet, van Antwerpen, et al., 2020a). This specification focuses on name binding aspects of Java, and implements packages, top-level and nested type definitions, type inheritance. Overloading is partly supported, while generics and lambda expressions are not supported.

We used three existing Java projects (`commons-{csv,io,lang3}`) and one generated Java project for the benchmark (`single-unit-clusters-call`). The existing Java projects are projects from the Apache Commons project that have no dependencies besides the Java standard library (JRE). The projects have different sizes, which allows us to asses the impact of project size on potential speedup. The generated project serves as a baseline for what is achievable with our parallel Statix implementation. It consists of a 100 classes, each class in its own package. The classes contain a number member methods, and each method body consists of method calls to other members of the same class. Because the classes are isolated and do not depend on other classes, the resulting compilation units only interact with the package's compilation unit and the unit for the Java standard library, and represent an ideal scenario in terms of parallelization. All projects and project sizes are listed in Figure 5.21.

Early experiments showed that the JRE, which is also treated as a compilation unit, often became the critical unit if it was served by a single actor. To eliminate this effect, the JRE is hosted on as many actors as the number of used cores, using round-robin scheduling to distribute queries over the actors. This is possible because the scope graph for the JRE is precomputed and statically loaded at the start of type checking.

We ran our type checker on each of these projects using an increasing number of cores. The benchmarks were executed using the JMH benchmark tool (OpenJDK, n.d.) in single-shot mode (the analysis was run once per iteration) using 5 warm-up and 15 measurement iterations. The benchmarks were executed on a Linux system with 128 AMD EPYC 7502 32-Core Processors 1.5GHz and 256GB RAM.

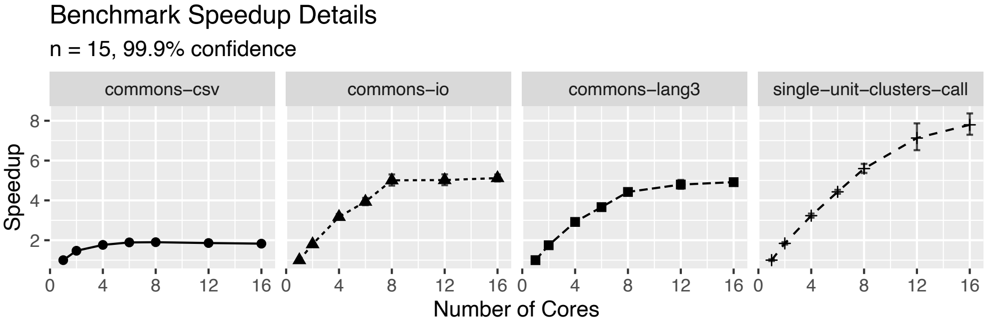


Figure 5.22: Benchmark results for type checking Java projects. Each subplot shows the speedup, relative to single-core speed, versus the number of used cores for each project. The benchmark was executed with 15 sample iterations, and the error bars represent a 99.9% confidence interval.

The results, shown in Figure 5.22, show the speedup of the parallel type checker, relative to the single core case, for the number of used cores. The error bars indicate the 99.9% confidence interval.

First, we see that the generated baseline project scales up to 5.6x for 8 cores. The scaling slows down more cores are used, but keeps increasing to ~7.8x for 16 cores.

Second, we see that the other projects all have a cut-off point after which adding more cores does not result in much speedup. The cut-offs are approximately at 4 cores for `commons-csv`, the smallest of the three, with a speedup of 1.8x, at 8 cores for `commons-io`, with a speedup of 5.0x, and at 8 cores for `commons-lang3`, with a speedup of 4.42x.

The cut-off in scaling can be explained by looking at the run time of individual compilation units. All projects contain a few source files that are significantly larger than most others. The cut-off happens when the run time of the whole problem is dominated by the run time of the largest compilation unit. If we look at the speedups discussed before, the run time of the longest-running unit as a percentage of the total run time was 84% for `commons-csv`, 100% for `commons-io`, and 81% for `commons-lang3`. Understanding why scaling slows down for some projects before the longest-running unit *completely* dominates the run time is an interesting question for future research.

These results suggest that our approach can give significant speedups for the Statix type checker. How well the approach scales depends on the type checker implementation as well as the granularity of parallelism. Our choice to parallelize on files means that the distribution of file sizes is important for the speedup that can be achieved. A type checker that supports more fine-grained parallelization (e.g., on method bodies), could possibly scale further. Our framework does not require file granularity and supports more fine-grained parallelism. Thus, users can experiment with the granularity that works well for their target language.

Note that these results are for a single type checker and for a single programming language. Both the implementation of the type checker and the design of the language may influence the possibility for effective parallel execution. The relation between language design, the resulting dependency patterns between compilation units, and the opportunity for parallelization is an interesting topic for future research. Our framework enables such experiments with parallel type checkers, by taking the hard parts of parallelization away from the compiler writer.

5.6.2 Supporting Local Inference

The Statix solver uses unification, and often relies on unification variables in scope graph data to be able to do inference. This posed a challenge when porting Statix to our framework. Our framework operates under the assumption that compilation units only communicate via the scope graph. This means the unifier of one compilation unit is not accessible to other compilation units. While the owning compilation unit can interpret that data relative to the local unifier, other units can not. We have a situation where a unit requires an incomplete view of its own data, but other units should only ever get the complete data.

We added a small extension to the framework to support such local inference patterns. Type checkers can define a function that produces a representation of data that is fit for other units:

```
async function GetExternalRepresentation(d)
```

The function is asynchronous so the type checker can delay returning the external representation until unification variables are instantiated. It is applied to the data of any scope whose owner is not the unit that issued the query. This solution allows units to do local type inference via the scope graph, while still presenting complete data to other units.

In the Statix literature, different patterns are used to associate declarations with types. In the first, the declaration and type are combined as a tuple $(x : T)$, and stored as the data in a single scope (van Antwerpen et al., 2018). In the second, the declaration only carries the name as data, and the type is represented as the data of a separate scope connected to the declaration by an edge (Rouvoet, van Antwerpen, et al., 2020a). We observed that the first encoding quickly results in deadlock if name resolution queries (resolve x) are necessary to instantiate the types T: The external representation of the whole tuple gets stuck on the logical variables in the type, therefore blocking the query for the name. The representation using tuples can easily be converted into the latter, but is a necessary consequence of the isolated nature of the compilation units in our approach.

5.7 Related Work

In this section we discuss related work on parallel approaches to build systems, compilers, and program models used for compiler and static analysis implementation.

5.7.1 Parallel Compilers

Parallel compilers are certainly not a given, even for often used languages, but there are several languages for which parallel compilers (mature or research prototypes) exist. These compilers are all for specific languages, but it is interesting discuss the techniques they use or the performance results they achieve. Although it is hard to find reliable information on the parallel capabilities of compilers, online discussion in StackExchange suggest that compilers for at least Java, C/C++, and C#, all often used, are all single-threaded (StackExchange, n.d.[a],[b],[c]). The concurrent compiler for Active Oberon (Reali, 2000) implements ideas that are similar to ours. Scopes (following the program nesting structure) have an associated state describing whether all symbols in the scope have been defined, and queries are delayed if scope information is incomplete. The supported scoping structure is specific for the target language and deadlock is avoided by being careful about what queries are done in what compilation phase. The implementation uses a shared data structure for the symbol table with a global lock, which is different from our approach of a distributed scope graph and units communicating by messages only. Hydra (Triplequote, n.d.) is a commercial parallel compiler for Scala, which parallelizes the Scala compiler by running the many phases of the Scala compiler in parallel. Hydra publishes benchmark results and reports speedups between 1.8–3.5x, depending on the project, on 4 cores (Triplequote, n.d.). Work has been done to parallelize the Rust compiler (Rust, n.d.). The approach is focused in parallelizing loops in the compiler, while maintaining most of the current structure of the compiler. However, at the time of writing the documentation mentions that "work on explicitly parallelizing the compiler has stalled. There is a lot of design and correctness work that needs to be done." The Go compiler supports parallel compilation at certain levels of the program (Go, n.d.). Particularly, the compilation of functions inside a package is executed in parallel. Finally, the Swift compiler takes an interesting approach to achieving parallel build (Swift, n.d.). Every compilation task has a focus, the compilation unit it "really" needs to compile. In the process it also compiles other units, but only as much as necessary for the focus unit. They claim this generally works well, because the necessary work on other units is limited.

5.7.2 Parallel Build Systems

Our framework shares many characteristics with build systems, as they run and order compilation tasks based on a dependency graph. The well known build tool

Make (Stallman et al., 2016) executes build tasks based on a statically known acyclic dependency graph. When the object language allows separate compilation, it can run these tasks in parallel as well. Many other build tools follow the model of Make, and require the dependencies to be acyclic and known a-priori. Some build tools such as Pluto (Erdweg, Lichter, et al., 2015) and PIE (Konat et al., 2018) improve on this model by supporting dynamic dependency discovery. The resulting dependency graph is still required to be acyclic. What all these have in common is that the build tasks are all treated as atomic operations, producing outputs from inputs. The build system is concerned with ordering these tasks correctly. This is in contrast with our approach, which makes partial results of a unit available to other units before it is completely finished. This allows us to support not only dynamic dependencies, but also cycles in the dependency graph, something that build systems cannot handle.

5.7.3 Parallel Programming Models

Another approach is to write the compiler in a programming model that supports parallel execution, and the parallelization is not organized around compilation units anymore.

The JastAdd framework for reference attribute grammars supports implicitly parallel attribute evaluation (Öqvist and Hedin, 2017). The resulting concurrency is more fine-grained than in our approach, and not necessarily driven by dependencies between compilation units. If one writes a compiler using reference attribute grammars, this is a convenient way to parallelize the compiler. Compared to our approach, reference attribute grammars do not provide a ready to use model for name binding. This means it falls on the developer to come up with suitable representations and algorithms for the object language. Applying parallel attribute evaluation to the ExtendJ Java compiler resulted in speedups of 1.52–2.43x on 4 cores. Although their evaluation was done on a different set of Java projects, these results suggest that the performance of our approach is competitive.

LVish (Kuper et al., 2014) proposes a parallel programming model based on monotonically growing data and freezing variables that reached a final state to achieve quasi-deterministic parallelism. It has been successfully applied to parallel type inference (Newton et al., 2016). This model is very similar to how we handle scope states: closing scopes and edges corresponds to freezing. The difference is that LVish is only a model for monotone state, which leaves users to build parallelization around it. The scope state model is specialized for our purpose, which allows us to make the parallelization and deadlock handling implicit for the user.

The theorem prover Isabelle/PIDE has strong support for implicit parallelization of proof checking (Matthews and Wenzel, 2010; Wenzel, 2009, 2013). The granularity is much smaller than in our approach. They do not support cyclic dependencies between parallel task, but a high degree of parallelism is achieved by exploiting proof irrelevance: most dependencies are only on the level of the theorem statements, but not their proofs. They report speedups up to 5.2–6.4x on 8 cores (Wenzel, 2013).

Several parallel programming models have been developed targeting static analyses. Because of their focus, these approaches target certain kinds of computations that are commonly used in static analyses, such as fixed points over lattices (Helm et al., 2020), parallel iteration over sets (Kulkarni et al., 2007), or established algorithms such as IDFS (Rodriguez and Lhoták, 2011).

5.7.4 Scope Graphs

Scope graphs (Néron et al., 2015) were introduced as a language-independent model of name binding with a focus on expressive, non-lexical, binding patterns, formalizing and generalizing the semantics of NaBL (Konat et al., 2012). This model was subsequently extended and used to develop formalisms for the specification of type checkers (van Antwerpen et al., 2018; van Antwerpen et al., 2016), resulting in the meta-language Statix. Followup work (Rouvoet, van Antwerpen, et al., 2020a) defined a formal, non-parallel, operational semantics for Statix, and proved it correct. It introduced the notion of *critical edges* as a tool to reason about query answer correctness in evolving scope graphs. Critical edges were defined in terms of the constructs of the Statix language and the presented operational semantics. In this paper we introduce *scope state* as an explicit and application independent description of the state and transitions of a scope in a evolving scope graphs, which was only implicitly present in the Statix operational semantics. Porting Statix to the parallel framework of this paper required reformulating the safety conditions of the original operational semantics to explicit scope state operations. All this work has been developed and applied in the context of the Spoofax language workbench (Kats and Visser, 2010).

5.8 Conclusion

In this paper we have introduced a framework for the implementation of implicitly parallel type checkers. We have introduced the concept of scope state to make the notion of weakly critical edges in evolving scope graphs explicit. We have presented a case study and shown that the approach does result in speedups for the larger projects in our benchmark. For all real-world projects in the benchmark the scaling was limited by a few large files, which suggest that more fine-grained parallelism (e.g., checking method bodies in parallel) could improve parallelism for this Java type checker. In general, investigating the relation between the type checker implementation/Statix specification and the achievable parallelization for different target languages is interesting follow-up research. Other interesting directions for future research are (a) extending this work to *incremental* type checking of large software projects during development, (b) developing useful abstractions for managing scope state that sit between the fine-grained API of this paper, where scope state is completely explicit, and the high-level abstraction offered by the Statix meta-language, where scope state and evaluation order are completely implicit, and (c) investigating how this work can be extended to and/or integrated with other compiler tasks such as parsing, and code generation to create a fully parallelized compiler pipeline.

Acknowledgments We thank the anonymous reviewers for their helpful comments. This research was funded by the NWO VICI Language Designer's Workbench project (639.023.206).

Towards Language-Parametric Semantic Editor Services

6

Abstract Editor services assist programmers to more effectively write and comprehend code. Implementing editor services correctly is not trivial. This paper focuses on the specification of semantic editor services, those that use the semantic model of a program. The specification of refactorings is a common subject of study, but many other semantic editor services have received little attention. We propose a language-parametric approach to the definition of semantic editor services, using a declarative specification of the static semantics of the programming language, and constraint solving. Editor services are specified as constraint problems, and language specifications are used to ensure correctness. We describe our approach for the following semantic editor services: reference resolution, find usages, goto subclasses, code completion, and the extract definition refactoring. We do this in the context of Statix, a constraint language for the specification of type systems. We investigate the specification of editor services in terms of Statix constraints, and the requirements these impose on a suitable solver.

6.1 Introduction

Editor services, such as syntax highlighting, reference navigation, and variable renaming, are an important tool for programmers. For example, code navigation is important for effective comprehension of code (Robillard et al., 2004), and refactoring approaches rely heavily on good tool support (Mens et al., 2003). It is therefore no surprise that such services are regularly used by users of IDEs (Murphy et al., 2006).

Editor services can be classified into syntactic and semantic editor services. The former, such as syntax highlighting, rely only on the (abstract) syntax of a program. The latter depend on a semantic model of the program, and use type or name binding information. Semantic editor services can be further divided in two groups: services that *inform* about the program, and services that *transform* the program. Informing services depend on the program model that is the result of type checking. The program model contains information on the types of variables, the declarations that ref-

Published as Daniel A. A. Pelsmaeker, Hendrik van Antwerpen, and Eelco Visser (2019). "Towards Language-Parametric Semantic Editor Services Based on Declarative Type System Specifications (Brave New Idea Paper)." In: *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*. Dagstuhl, Germany. DOI: 10.4230/LIPIcs.ECOOP.2019.26.

erences refer to, etc. Transforming services are guided by the program model (e.g., to rename a declaration and all its usages), but may also rely on the typing rules to ensure the resulting, transformed program is well-formed.

Implementing semantic editor services and ensuring their correctness is not a trivial task (see, e.g., the difficulties around correctly implementing Java refactorings; M. Schäfer and de Moor, 2010). Language workbenches are tools to aid the development of programming languages and programming environments (Fowler, 2005) by means of declarative formalisms and reusable tools that support correctness and reduce development effort. A good example of this is the use of a context-free grammar to specify syntax. This specification can be used to drive a parser, but also for unparsing, or to provide syntactic code completion. The language developer writes a declarative specification, which helps with the correctness of the syntax, while existing parsing, unparsing, and code completion algorithms can be reused, reducing development time.

However, even though "editor support is a central pillar of language workbenches" (Erdweg et al., 2013), and many language workbenches do indeed support many common editor services, there is little literature on reusable formalisms and algorithms for their definition (Omar, Voysey, Hilton, Sunshine, et al., 2017). An important exception is the extensive work on defining correct refactorings (e.g., M. Schäfer and de Moor, 2010; Steimann, 2018; Tip et al., 2011). However, many editor services common to modern IDEs, such as reference resolution, finding declaration usages, or semantic code completion, have received little attention.

In this paper we argue that a range of semantic editor services, beyond those that have already appeared in the literature, can be specified as constraint problems. Constraints separate the declarative specification of a problem from the operational interpretations necessary to solve it. This separates concerns, but also allows reuse of constraint-based specifications for different purposes. For example, in addition to verifying the correctness of the static semantics of a program, constraint-based typing rules have also been successfully used in the implementation of semantically correct refactorings (Steimann, 2018).

Many editor services rely on name binding information, where complex scoping and name binding rules can be a challenge for the correct implementation of editor services (e.g., correct Java refactorings involving names; M. Schäfer et al., 2012). Although constraint-based formulations of typing rules are pervasive, constraint-based formulations of the scoping and name binding rules are rare. Name binding introduces complexities, such as avoiding accidental name capture when refactorings introduce new names. We believe that treating name binding and name resolution as an integral part of the constraint problem increases the applicability of a constraint-based approach to editor services, and can improve existing specifications from the literature in this regard.

As the basis for our investigations we use Statix, a constraint language developed for the specification of type systems (van Antwerpen et al., 2018). Statix is built around scope graphs, a language-independent model for name binding and name resolution (van Antwerpen et al., 2016). We argue that Statix is a suitable basis for the definition of editor services by expressing them in terms of Statix constraints and Statix type system specifications. Although Statix constraints are suitable for a declarative specification of editor services, the current deterministic solver algorithm of Statix, suitable for type checking and code navigation, is not capable of solving the editor scenarios we discuss. We identify requirements for an alternative solver for Statix that does support the interpretation and solving algorithms required for our proposed editor service definitions.

Specifically, we have the following contributions:

- We express several common editor services in terms of Statix constraint problems.
- We identify requirements on an operational semantics of Statix that is able to solve these problems.

This paper is organized as follows. Section 6.2 discusses the characteristics of semantic editor services and motivates our choice of editor services. In Section 6.3, we introduce Statix, and Statix type specifications using an example. In Section 6.4 we express several informing editor services in terms of the resulting program model. In Section 6.5 and Section 6.6, we do the same for the semantic code completion and extract definition refactoring editor services, respectively. In Section 6.7 we discuss related work. We conclude and discuss future work necessary to fully realize our proposed approach in Section 6.8.

6.2 Characterizing Editor Services

Editor services can be characterized as syntactic, those that only need the syntactic model of the program to work, and semantic, those that require the semantic model of the program (Erdweg et al., 2013). We can further distinguish the semantic editor services by whether they transform the program, or merely inform the user. The informing services include editor services such as goto declaration, finding and highlighting usages, navigating to the supertype, and listing all overriding methods. The transforming editor services include quick fixes, static semantics-preserving refactorings, and (semantic) code completion.

In this section we discuss aspects that distinguish the various semantic editor services, and motivate our choice for the editor services we discuss.

Completeness Some editor services have to be able to work on syntactically and/or semantically incomplete programs. For example, as code completion can be invoked while the user is typing, it must be able to deal with a program that is both syntactically and semantically incomplete. Similarly, the *fix import* quick fix that adds an import statement to a program to make a reference resolve, must be able to deal with programs that have incomplete semantic information; namely the program with the reference that initially does not resolve. Other editor services could provide a better user experience if they can deal with syntactically or semantically incomplete programs, but this is not a requirement.

Preserving Static Semantics The transforming editor services all need to preserve the existing semantics of the program up to some degree. Refactorings such as *rename refactoring* and *extract definition* tend to have very strict semantic preservation requirements, including that all existing references need to resolve to the same declarations before and after the refactoring. Quick fixes and code completion, by their nature, introduce new syntax that may change certain local semantics of the code, but should not have an impact outside the area of influence.

Concrete Name Generation Often, transforming editor services add new declarations to the program as part of their refactoring or fixing behavior. These declarations need a concrete name, one which is syntactically valid and does not clash with existing names in the program. That is, the new name should not overlap with existing names, or cause inadvertent variable capture.

What We Study Given the characterization above, we picked five editor services for which we describe the ideas of this paper. As informing editor services we choose *reference resolution*, *find usages*, and *list subclasses*, because they show how scope graphs can be used to answer these queries, where the last one requires language-specific knowledge. The last two also explore how flexible the solver must be to be able to answer such inverse queries.

We discuss two transforming editor services: *code completion*, which will have to deal with syntactically incomplete programs, and the *extract definition* refactoring, which is interesting because it introduces new syntax for which we want to use the solver to find the concrete, semantically correct, values to fill in.

We do not claim that these editor services cover all issues, or that the resulting requirements cover all editor services. However, we think that they exhibit a sufficient range of features to show the range of possibilities, and expose important requirements that need to be fulfilled to realize our approach.

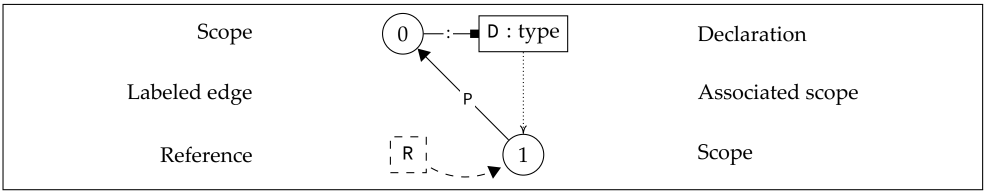


Figure 6.1: Overview of the notation used for scope graphs in this paper.

6.3 Introduction to Statix

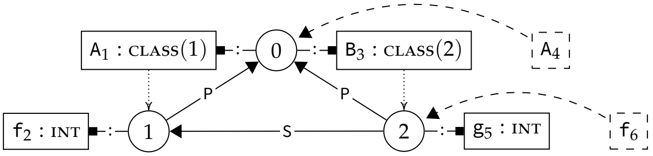
Statix is a recently introduced meta-language for the specification of static semantics (van Antwerpen et al., 2018), based on scope graphs and constraints (Néron et al., 2015; van Antwerpen et al., 2016). We chose Statix because it allows us to declare semantic editor services in terms of constraints and type system specifications.

First, we explain scope graphs, a language-independent model for name binding and name resolution. Then, we introduce the rules for static semantics, and their (declarative) meaning. Finally, we explain how type checking based on these rules is implemented. We use the Java program in Figure 6.2 as a running example. The subscripts on program identifiers are a notational convention we use to distinguish different occurrences of the same name.

Name Binding with Scope Graphs In Statix the name binding and resolution is part of the constraint problem, to allow complex interactions between type checking and name resolution. The name binding structure of a program is represented as a language-independent model called a *scope graph* (Néron et al., 2015; van Antwerpen et al., 2018; van Antwerpen et al., 2016), which is a graph of scopes and declarations in those scopes. As shown in Figure 6.1, the scopes are connected by labeled, directed edges. Name resolution corresponds to a query finding a path in the graph to a matching declaration.

Consider our example program and the corresponding scope graph in Figure 6.2. The global scope of the whole program is represented by the circled node 0. The definition of class A corresponds to a declaration $\mathtt{A}_1$. Declarations contain both the name and its type, and therefore use the *'is of type'*-symbol ":" to label these edges. Class types are represented by the class scope. For example, scope 1 is the scope of class A, and its type is $\textsc{class}(1)$. The class scopes are lexical sub-scopes of the global scope, which is modeled by the P-labeled (parent) edges. The fact that class B extends class A is represented by the edge labeled S (supertype). This edge makes the fields from the super class visible in the subclass, but is also used to decide subtyping between class types. The field declarations are similar to the class declarations, but in the class scopes.

```
class A1 {
    int f2 = -1;
}
class B3 extends A4 {
    int g5 = f6;
}
```



$$\text{(J-ClassDec)}\ \dfrac{\begin{array}{c}\nabla s_c \quad s_c \xrightarrow{\mathsf{P}} s \quad s \xrightarrow{:} C_i : \textsc{class}(s_c) \quad s_c \vdash \overline{d}\ \textsc{ok} \\ \mathsf{query}\ s \xmapsto{\mathsf{P}^*:} \textsc{decl}(D_j)\ \mathsf{as}\ D_k : \textsc{class}(s_d) \quad s_c \xrightarrow{\mathsf{S}} s_d\end{array}}{s \vdash \texttt{class}\ C_i\ \texttt{extends}\ D_j\ \{\ \overline{d}\ \}\ \textsc{ok}}$$

$$\text{(J-FieldDec)}\ \dfrac{\begin{array}{c}s \vdash [\![t]\!] \Rightarrow T \quad s \xrightarrow{:} f_j : T \\ s \vdash e : T' \qquad \vdash T' <: T\end{array}}{s \vdash t\ f_j = e;\ \textsc{ok}} \qquad \text{(J-Var)}\ \dfrac{\mathsf{query}\ s \xmapsto{\mathsf{P}^*\mathsf{S}^*:} \textsc{decl}(x_i)\ \mathsf{as}\ x_j : T}{s \vdash x_i : T}$$

$$\text{(T-Int)}\ \dfrac{}{s \vdash [\![\texttt{int}]\!] \Rightarrow \textsc{int}} \qquad \text{(T-Class)}\ \dfrac{\mathsf{query}\ s \xmapsto{\mathsf{P}^*:} \textsc{decl}(C_i)\ \mathsf{as}\ C_j : T}{s \vdash [\![C_i]\!] \Rightarrow T}$$

$$\text{(<:-Int)}\ \dfrac{}{\vdash \textsc{int} <: \textsc{int}} \qquad \text{(<:-Class)}\ \dfrac{\mathsf{query}\ s_1 \xmapsto{\mathsf{S}^*} \textsc{scope}(s_2)\ \mathsf{as}\ p}{\vdash \textsc{class}(s_1) <: \textsc{class}(s_2)}$$

Figure 6.2: Example Java program with two classes, its corresponding scope graph, and the relevant Statix typing rules.

Resolving a name corresponds to querying the scope graph for a matching declaration. Resolution queries are parameterized by a regular expression that determines which declaration can be reached, a predicate determining which declarations match. An additional order on labels is used to disambiguate multiple matching declarations. For example, the class reference A_4 is resolved in the global scope 0. Class references are resolved in the lexical context, and the regular expression that encodes this is P^*:, which matches any path to a declaration via any number of P-steps to lexical parents. The declaration itself should match the reference, which is specified with the predicate $\textsc{decl}(\mathsf{A}_4)$, which holds for any x_i where $x = \mathsf{A}$. In this case the reference resolves directly to declaration A_1 in scope 0.

Resolving the variable reference f_6 follows the same pattern. However, it should be possible to resolve not just to variables in the lexical context, but also to fields in the super class. This is achieved by using the regular expression $\mathsf{P}^*\mathsf{S}^*\mathsf{:}$. This allows the reference to be resolved to declaration $\mathtt{f}_2$, by following the S-edge to scope 1.

Type Specifications The rules of a Statix specification formally describe the scope graph that corresponds to a program, as well as constraints on references and types, in terms of syntax-directed rules. Figure 6.2 shows some of the rules that apply to our example program. For example, the rule (J-ClassDec) specifies that a class definition c is well-formed in scope s, written as $s \vdash c$ $\textsc{ok}$, if the scope graph has the correct structure, and the definitions in the class are well-formed as well ($s_c \vdash \overline{d}$ $\textsc{ok}$). The first three premises state that the scope graph contains a scope s_c that is unique to this class (∇s_c), that this scope has a P-edge to its lexical parent ($s_c \xrightarrow{\mathsf{P}} s$), and that there is a declaration C_i for the class in the lexical scope s, typed by the class scope s_c ($s \xrightarrow{:}\!\blacksquare\, C_i : \textsc{class}(s_c)$). The last two premises say that the reference to the super class resolves to a declaration D_k, which is typed by a class scope s_d (query $s \xmapsto{\mathsf{P}^*\mathsf{:}}$ $\textsc{decl}(D_j)$ as $D_k : \textsc{class}(s_d)$), and that an inheritance edge exists from the scope of this class to the scope of the super class ($s_c \xrightarrow{\mathsf{S}} s_d$).

The rule (J-FieldDec) specifies that a field declaration is well-formed if a declaration for the field exists in the scope graph ($s \xrightarrow{:}\!\blacksquare\, f_j : T$), if the assigned expression is well-typed for some type T' ($s \vdash e : T'$), and the expression type T' is a subtype ($\vdash T' <: T$) of the semantic type T corresponding to the type annotation ($s \vdash [\![t]\!] \Rightarrow T$). The relations for semantic typing, subtyping, and expression typing are also defined with Statix rules. The only built-in constraints are constraints to define the scope graph, constrains to query the scope graph, and term equality. All other relations are completely determined by the rules from the specification.

Type Checking The specification is declarative, and only gives a logical description of what well-formed programs are with respect to a scope graph. We made no assumptions yet on how to operationalize it. One possible interpretation is to use the specification to type check programs. Checking that a program p is well-formed corresponds to checking if the constraint $s \vdash p$ $\textsc{ok}$ is satisfiable. Van Antwerpen et al. describe an algorithm to solve such constraints, given a specification and a program p as input (van Antwerpen et al., 2018). The algorithm uses the rules from the specification to simplify constraints until only built-in constraints remain. These are solved using unification and scope graph resolution algorithms. This solver is deterministic: it does not use back-tracking, and only applies rules if they match the given program construct. The result of solving a constraint such as $s \vdash p$ $\textsc{ok}$ is a solution consisting of a variable assignment V and a scope graph G, or no solution if the constraint cannot be satisfied. A resulting program model would also include the types assigned to all expressions, and the resolution R of all references in the program.

6.4 Informing Editor Services

Many editors have editor services through which the user can navigate their program. The simplest of these involve clicking a reference and jumping to the corresponding declaration, or listing all usages of a declaration, but there are also more sophisticated editor services such as those that list the subclasses of a particular class. All these services have in common that they can be expressed as queries on the program model that resulted from type checking. Even though these queries themselves do not change the program, they may be part of the implementation of other editor services that do change the program. For example, a refactoring that renames a variable first needs to find all usages of the variable to ensure they are all renamed.

Reference resolution and finding declaration usages can easily be derived from the program model, which contains the resolution relation R, which consists of pairs of references and their declaration. Consider the example in Figure 6.2 again. Finding the declaration corresponding to reference A_4, involves finding the entry for the reference in R. Conversely, finding all usages of declaration $\mathtt{f}_2$ corresponds to a reverse lookup. These queries parallel the resolution queries in the typing rules, and can directly be derived from the specification.

While a query to find all subclasses of a certain class is not directly present in the typing rules, we can phrase such a query as a constraint, which we solve with respect to the given program model. For instance, how would we specify — in constraints — the query to get all subclasses of class A_1? We assume as input the declaration itself, and the scope 0 of the class definition, which should be part of the program model. The general idea of the query is to find the class scope, find other class scopes that are connected to it by inheritance edges, and find their corresponding declarations. This is encoded by the following constraint:

$$\mathsf{query}\ 0 \overset{\vdots}{\mapsto} \textsc{decl}(\mathsf{A}_1)\ \mathsf{as}\ \mathsf{A}_1 : \textsc{class}(s_c)$$

$$\mathsf{query}\ s_d \xmapsto{\mathsf{S}^+} \textsc{scope}(s_c)$$

$$\mathsf{query}\ s' \overset{\vdots}{\mapsto} \textsc{true}\ \mathsf{as}\ x_i : \textsc{class}(s_d)$$

where s_c, s_d and s' are existentially quantified, and x_i is the output. The first constraint says that the class declaration is typed by a scope s_c. The second constraint states that there is a path from some subclass scope s_d to the class scope s_c. The final constraint indicates that there is a declaration with any name x_i, which is typed by the subclass scope s_d.

None of these constraints appear as such in the typing rules, and we have to do work to find possible solutions. This may seem daunting, given the free variables for scopes and names, both of which have infinite domains. However, we are only interested in solutions that are valid in the context of an existing scope graph. This scope graph is always finite, which gives us an initial, if maybe inefficient, strategy to find possible solutions. In the case of our example, there is one solution, where $s_c = 1$, $s_d = 2$, $s' = 0$, and $x_i = \mathsf{B3}$.

The given formulation requires an algorithm quite different from the current, deterministic solver of Statix. Instead of strictly relying on inference via forward resolution and unification, it needs to be able to guess values, try different alternatives, and back-track on failed attempts. An alternative approach could have been to change scope graph queries to allow backward edge steps. For example, if we use $\hat{l}$ for backward steps in the regular expression, our second constraint might have been:

$$\mathsf{query}\ s_c \xmapsto{\hat{\mathsf{S}}+} \textsc{true}\ \mathsf{as}\ s_d$$

In this case, we could do forward resolution from scope s_c again, reusing the resolution algorithm that is already there. Although this approach may work for queries designed specifically with editor services in mind, it does not work if we want to use our typing rules as-is. Therefore, we choose not to change the formalism, but require a solver that supports more flexible inference.

Summary We showed that queries on the program model can be expressed as constraints, and that finding answers to these queries corresponds to solving these constraints in the context of a given program model. We discussed that solving these queries requires different solver strategies to be supported by the solver for Statix. However, this solver would be independent of the specific object language the query is for, and is therefore reusable between languages. Given such a solver, implementing such editor services reduces to being able to specify the query as a constraint.

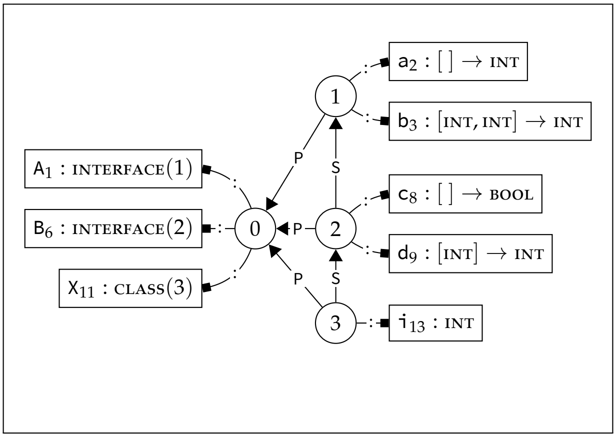
6.5 Code Completion

Code completion is an editor service that suggests a valid code fragment to be inserted at the caret position. This assists the user while typing, attempts to minimize typing errors, and aids in discovery by showing the possible syntax and references. Syntactic code completion is the most basic kind of code completion: it suggests only syntax fragments that fit at the caret location, with no regard for whether the proposal fits semantically. Semantic code completion improves on this by suggesting only those proposals that conform to the static semantics of the language, such as only suggesting expression syntax that can produce a value of the expected type. Additionally, semantic code completion proposes inserting references to declarations, such as variables, fields, and functions, that are visible from the scope at the caret location. In this section we discuss how the type system and semantic specification of a language can be used to provide accurate semantic code completion without additional work on the part of the language designer.

```
interface A1 {
  int a2();
  int b3(int x4, int y5);
}

interface B6 extends A7 {
  boolean c8();
  int d9(int x10);
}

class X11 implements B12 {
  int i13 = |;
}
```



$$\text{(J-InterfaceDec)}\ \frac{\begin{array}{c}\nabla s_c \quad s_c \xrightarrow{\mathsf{P}} s \quad s \xrightarrow{:} C_i : \mathrm{INTERFACE}(s_c)\\ \mathsf{query}\ s \mapsto \mathrm{DECL}(\overline{D_j})\ \mathsf{as}\ \{\overline{D_k} : \mathrm{INTERFACE}(\overline{s_d})\}\\ s_c \xrightarrow{\mathsf{S}} \overline{s_d} \qquad\qquad s_c \vdash \overline{d}\ \mathrm{OK}\end{array}}{s \vdash \texttt{interface}\ C_i\ \texttt{extends}\ \overline{D_j}\ \{\ \overline{d}\ \}\ \mathrm{OK}}$$

$$\text{(J-InterfaceMethodDec)}\ \frac{s \vdash [\![C_i]\!] \Rightarrow T \quad s \vdash [\![\overline{C_k}]\!] \Rightarrow \overline{T_k} \quad s \xrightarrow{:} x_j : \overline{T_k} \to T}{s \vdash C_i\ x_j(\overline{C_k\ x_k});\mathrm{OK}}$$

$$\text{(J-ClassDec)}\ \frac{\begin{array}{c}\nabla s_c \quad s_c \xrightarrow{\mathsf{P}} s \quad s \xrightarrow{:} C_i : \mathrm{CLASS}(s_c) \quad s_c \vdash \overline{d}\ \mathrm{OK}\\ \mathsf{query}\ s \mapsto \mathrm{DECL}(\overline{D_j})\ \mathsf{as}\ \{\overline{D_k} : \mathrm{INTERFACE}(\overline{s_d})\} \quad s_c \xrightarrow{\mathsf{S}} \overline{s_d}\end{array}}{s \vdash \texttt{class}\ C_i\ \texttt{implements}\ \overline{D_j}\ \{\ \overline{d}\ \}\ \mathrm{OK}}$$

$$\text{(J-FieldDec)}\ \frac{\begin{array}{c}s \vdash [\![t]\!] \Rightarrow T \quad s \xrightarrow{:} f_j : T\\ s \vdash e : T' \qquad \vdash T' <: T\end{array}}{s \vdash t\ f_j = e;\ \mathrm{OK}}$$

$$\text{(J-ThisMethodCall)}\ \frac{\begin{array}{c}s \vdash \overline{e} : \overline{V} \qquad\qquad \vdash \overline{V} <: \overline{U}\\ \mathsf{query}\ s \overset{\mathsf{S}^*}{\longmapsto} \mathrm{DECL}(m_i)\ \mathsf{as}\ \{m_j : \overline{U} \to T\}\end{array}}{s \vdash m_i(\overline{e}) : T}$$

$$\text{(J-Plus)}\ \frac{s \vdash e_1 : T_1 \quad s \vdash e_2 : T_2 \quad T_1 = T_2 = T = \mathrm{INT}}{s \vdash e_1 + e_2 : T}$$

Figure 6.3: Java program illustrating code completion, and the corresponding scope graph and relevant Statix typing rules.

In Figure 6.3 we show an example Java program with the caret position denoted by |, near the end of the last line of class `X`$_{11}$. The program is incomplete: it is not syntactically valid because the user has not yet finished typing. Despite this, we would want the semantic model of the program so we can suggest relevant syntax and references.

As a first step, we propose to use the techniques described by de Souza Amorim et al. (2016) to use the syntactic specification of the language to introduce *placeholders* into the abstract syntax. A placeholder is a term in the syntax that represents a place where syntax of a certain sort, such as an expression or a declaration, could be inserted. This makes the program syntactically complete, and the placeholders provide us with syntax terms which we can constrain. Therefore, to the completion service, the incomplete line of code has the following syntax, with placeholder `$Exp` for a possible expression that would complete the program:

$$\texttt{int i}_{13}\texttt{ = }\mathit{\$Exp}\texttt{;}$$

At this point, we would want to invoke the solver and let it verify our program using the rules shown in Figure 6.3. However, no rules apply to the placeholder term `$Exp`. Instead, we propose to replace any occurrence of a placeholder in the syntax terms with a corresponding constraint variable in the constraint terms. In this example, we use ε for `$Exp`, which, because of the semantic rule (J-FieldDec), results in the following constraints for this line:

$$3 \vdash [\![\texttt{int}]\!] \Rightarrow T \qquad 3 \overset{\vdots}{\longrightarrow}\!\blacksquare\ \texttt{i}_{13} : T \qquad 3 \vdash \varepsilon : T' \qquad \vdash T' <: T$$

Solving these constraints assigns $T' \mapsto \textsc{int}$ and $T \mapsto \textsc{int}$. In other words, the editor service has inferred that the expected type of the expression on that line must be $\textsc{int}$, and produced the scope graph shown in Figure 6.3. The solver can continue, trying to find an assignment for ε. There are two rules in Figure 6.3 that it could apply: (J-Plus) and (J-ThisMethodCall). In fact, we would want the solver to return both solutions for code completion. We will explore both these alternatives.

Expression Completion From rule (J-Plus) ($s \vdash e_1 + e_2 : T$) we would get the assignment $\varepsilon = \varepsilon_1 + \varepsilon_2$, where ε_1 and ε_2 are new constraint variables introduced by the solver. We would like to stop here, and let the solver return the solution $\varepsilon = \varepsilon_1 + \varepsilon_2$. Note that this solution is incomplete: it does not describe the whole program as there are still free constraint variables in them. Therefore, the solver would need to be able to return incomplete solutions. As part of this solution, we get some constraints that not ground because they contain these free constraint variables:

$$3 \vdash \varepsilon_1 : \textsc{int} \qquad\qquad 3 \vdash \varepsilon_2 : \textsc{int}$$

Translated back to syntax terms, replacing the free constraint variables by placeholders, this would result in the following syntax on the line being completed:

```
int i13 = $Exp + $Exp;
```

Of course, we could also let the solver continue its search to find assignments for ε_1 and ε_2, but this would likely result in an ever expanding sequence of $\varepsilon_1 + \varepsilon_2 + \varepsilon_3 + \ldots$. Ultimately, there are infinitely many solutions if we were to try to make all variables ground. This shows that we need a way to instruct the solver on how deep we want a constraint variable to be solved. In this example, we want solutions for ε only one level deep.

Method Call Completion When the solver instead applies rule (J-ThisMethodCall), we get method call completion: where code completion suggests calls to methods in scope at the caret position, and whose return a type is compatible with the expected type of the expression. From rule (J-ThisMethodCall) ($s \vdash m_i(\bar{e}) : T$) we would get the assignment $\varepsilon = \mu(\bar{\varepsilon})$, again introducing new constraint variables μ and $\bar{\varepsilon}$ to represent the method name and arguments respectively.

Since proposing just the syntax for a method call is not very satisfactory to a user, this time we *do* want to get another level of solutions. At least, we want μ to be solved, but we do not care about $\bar{\varepsilon}$. We need a way to indicate this to the solver. Through the rule (J-ThisMethodCall) the solver would add these constraints:

$$3 \vdash \bar{\varepsilon} : \overline{V} \qquad \vdash \overline{V} <: \overline{U} \qquad \text{query } 3 \xmapsto{\mathsf{S}^*:} \textsc{decl}(\mu) \text{ as } \{m_j : \overline{U} \to \textsc{int}\}$$

There are multiple possible assignments for constraint variables μ and $\bar{\varepsilon}$, and for code completion to work, the solver must find them all. The following table shows the possible assignments for μ, $\bar{\varepsilon}$, T, $\overline{U}$, and $\overline{V}$ that the solver might yield.

Solution	μ	$\bar{\varepsilon}$	T	$\overline{U}$	$\overline{V}$
Solution 1	a_2	$[\,]$	INT	$[\,]$	$[\,]$
Solution 2	b_3	$[\varepsilon_1, \varepsilon_2]$	INT	[INT, INT]	$[\tau_1, \tau_2]$
~~Solution 3~~	c_8	$[\,]$	BOOL	$[\,]$	$[\,]$
Solution 4	d_9	$[\varepsilon_1]$	INT	[INT]	$[\tau_1]$

Note that solution 3 is not valid, as it tries to assign $T \mapsto$ BOOL whereas T had previously already been assigned INT. Also note how the solver could infer lists of constraint variables for $\overline{U}$ and $\overline{V}$. But, as before, we would not want the solver to keep expanding on the constraint variables it has introduced. If we had not relaxed these variables such that they may remain free, the solver would have to find some assignment for the variables that satisfies them. In this example the solver might have added a method call to an arbitrary method with a compatible return type, such as a_2. In other scenarios the solver may not be able to find such a solution, or find infinitely many.

The solutions returned by the solver can be turned into syntax fragments and presented to the user as code completion proposals, where we replace the free constraint variables by syntax placeholders. The order of the proposals is not determined by the solver, as we consider this to be a separate concern. For example, we may want to order the proposals by their frequency of use, or use the semantic model to order the proposals by closeness (e.g., local variables before global variables). In this example, code completion would propose the following method calls:

```
a2()
b3($Exp, $Exp)
d9($Exp)
```

Summary To use the semantic of the programming language for code completion, we first need a semantic specification that includes a model for name binding. This is already provided by the scope graphs used by the Statix constraint solver. However, the solver also needs to support returning incomplete solutions. The solver needs to be able to distinguish between constraint variables that we want to have solved and those that may remain free, and we need to be able to indicate how deep we want a given constraint variable to be solved. By using the semantic rules, a solution can include syntactic assignments to variables. Finally, the solver must be able to return more than one solution, so we can display them all to the user as part of code completion.

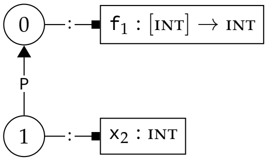
6.6 Extract Definition

A common refactoring is the *extract definition* refactoring, where the user selects a subexpression and the refactoring replaces any occurrences of that expression by a reference to a variable definition initialized by the subexpression. In the example in Figure 6.4, we want to extract the `x - 3` subexpression into a separate definition. We assume the program is syntactically complete and semantically correct.

The first step in this refactoring is to determine the new syntax that we expect as a result of the refactoring. This is language-specific syntax, selected by the user and specified in advance by the language developer. The syntax fragment uses placeholders, as shown below, where `$Type` is a placeholder for the type of the newly created variable and `$ID` is a placeholder for a variable name. In this case we want all three occurrences `$ID` to refer to the same variable.

```
int f1(int x2) {
    $Type $ID = x3 - 3;
    return $ID + $ID * x5;
}
```

```
int f₁(int x₂) {
    return (x₃ - 3) + (x₄ - 3) * x₅;
}
```



$$\text{(J-MethodDec)}\ \frac{\begin{array}{c} s \vdash [\![C_i]\!] \Rightarrow T \quad \nabla s_m \quad s_m \xrightarrow{\mathsf{P}} s \quad s \vdash [\![\overline{C_k}]\!] \Rightarrow \overline{T_k} \quad s_m \xrightarrow{:} \overline{p_k} : \overline{T_k} \\ s_m \vdash \overline{b}\ \textsc{ok} \quad s \xrightarrow{:} m_j : \overline{T_k} \to T \quad s_m \vdash e : U \quad \vdash U <: T \end{array}}{s \vdash C_i\ m_j(\overline{C_k}\ \overline{p_k})\ \{\ \overline{b}\ \texttt{return}\ e; \}\ \textsc{ok}}$$

$$\text{(J-VarDec)}\ \frac{s \vdash [\![C_x]\!] \Rightarrow T \quad s \xrightarrow{:} v_y : T \quad s \vdash e : T}{s \vdash C_x\ v_y = e;\ \textsc{ok}}$$

$$\text{(J-Var)}\ \frac{\mathsf{query}\ s \xmapsto{\mathsf{P^*S^*:}} \textsc{decl}(x_i)\ \mathsf{as}\ x_j : T}{s \vdash x_i : T}$$

$$\text{(J-IntBinOp)}\ \frac{s \vdash e_1 : T_1 \quad s \vdash e_2 : T_2 \quad T_1 = T_2 = T = \textsc{int}}{s \vdash e_1 \oplus e_2 : T}$$

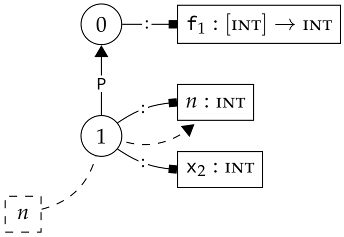
Figure 6.4: Java program before applying the *extract definition* refactoring, and the corresponding scope graph and relevant Statix typing rules.

We create a copy of the previous, valid, solution returned by the solver, and adapt it to this refactoring. This is a two-part process: relaxing the solution, and adding new constraints to the problem. Relaxing the solution removes any variables, resolutions, constraints, and scope graph nodes that are no longer valid or that impact the aspects we want to refactor. For extracting a definition, relaxation only involves removing the reference relation x₄ ↦ x₂, since the reference x₄ has been removed. However, we still want the variable references x₃ and x₅ to resolve to the same definition x₂.

Now we can add new constraints to the problem, but the refactoring should add only those constraints that result from the changed syntax. The constraints may contain syntax terms, but we replace any occurrences of the placeholders by constraint variables. For the type placeholder `$Type`, we will use the constraint variable τ. Since we want all occurrences of the `$ID` placeholder to refer to the same variable, we should replace all occurrences with the same constraint variable.

Where other approaches use the solver to find a concrete name for the variable (Steimann, 2018), we argue that this is not necessary for the solver to give a correct

```
int f(int x) {
    int n = x - 3;
    return n + n * x;
}
```



	n	τ	T	v_j
Solution	n	`int`	INT	n

Figure 6.5: Java program after applying the *extract definition* refactoring, and the corresponding scope graph and variable assignments. Note that n in the program is a rigid variable, which has yet to be assigned a concrete name.

result. We merely want to indicate to the solver that the new name is different from all other names in the program. This gives a separation of concerns: the solver can verify that the program satisfies the constraints without needing to produce any concrete names, and generating the concrete names can be done externally after the solver has verified the program. For example, some IDEs provide a list of name suggestions that they generate from the context, such as the type of the expression.

To distinguish the abstract name from any other name in the program, we can use a rigid variable: one that is distinct from any other variable or name. Similar to how rigid variables are used to create new distinct scopes in the scope graph (through ∇s), we create a new rigid variable to represent the name of the newly introduced variable: ∇n. Due to rules (J-VarDec) and (J-Var), this results in the following constraints:

$$\begin{array}{ll}
\nabla n & \\
1 \vdash [\![\tau]\!] \Rightarrow T & 1 \xrightarrow{:} n : T \\
1 \vdash (\mathsf{x}_3 - 3) : T & \text{query } 1 \xmapsto{\mathsf{P}^*\mathsf{S}^*:} \textsc{decl}(n) \text{ as } x_j : T
\end{array}$$

Solving these constraints results in the variable assignment and scope graph shown in Figure 6.5.

From this we can conclude that the refactoring is valid, does not semantically change the program (since the existing constraints and reference resolutions are preserved), and that the type of the newly introduced variable is `int`. However, to finish the refactoring we have to decide on a concrete name for rigid variable n. A concrete name can be provided by the refactoring tool or by the user. In any case, we can test whether the suggested name is allowed by reinvoking the solver with the new solution and one additional constraint: to constrain n to the chosen name, say `i`.

$$n = \texttt{i}$$

The new constraint may result in an invalid solution, for example when the chosen name overlaps with another, or causes inadvertent name capture somewhere in the program. In this example, `x` is not allowed as a concrete name for n. However, if this results in a valid solution, the concrete name is acceptable and the refactoring can finish. In this example it would produce the following code:

```
int f(int x) {
    int i = x - 3;
    return i + i * x;
}
```

Summary As part of the refactoring we generate new syntax, where we use placeholders to indicate where we need more information. In the newly generated constraints we have a constraint variable taking the place of every placeholder, which allows us to use the solver to find a solution to the problem. By using a rigid variable in place of a concrete name, we can indicate that the name is different from all other names in the program without having to specify such a name concretely, giving a separation of concerns between finding whether the program is valid and what concrete name to choose.

6.7 Related Work

Erdweg et al. (2013) identify editor services as an important aspect of language workbenches, and give an overview of commonly supported editor services. However, Omar et al. have pointed out that the study of the semantic foundations of editors and editor interactions has received little attention so far (Omar, Voysey, Hilton, Sunshine, et al., 2017). We discuss work related to code completion, and refactoring, as those are most relevant to the editor services we covered in this paper.

Reference Resolution Language workbenches such as Xtext (Eysholdt and Behrens, 2010) and Spoofax (Kats and Visser, 2010; Visser et al., 2014) provide support for language-parametric reference resolution based on declarative name binding specifications. Xtext supports specification of references in the language grammar as crosslinks, which specify the sort that an identifier can refer to. Xtext will check the validity of the references and add them to the model.

The first approach to declarative name binding specification in Spoofax was the NaBL name binding language (Konat et al., 2012). The name binding rules defined the definition sites and their scopes based on the abstract syntax of the program. The built-in reference resolution algorithm could only create an index in which references can be looked up, which limits its flexibility to be used in other editor services.

Instead, the approach we use in this paper uses the expressiveness of the constraints and the flexibility of the Statix constraint solver to enable reference resolution to be used in various editor services.

Code Completion The Xtext and Spoofax language workbenches also provide support for language-parametric syntactic completion, based on a syntax definition. In the case of Xtext, it suggests possible keywords. Spoofax suggests complete syntactic constructs, and represents incomplete syntax trees using placeholders that act like holes in the program text (de Souza Amorim et al., 2016). This representation is instrumental for translating an incomplete program to an abstract syntax tree with variables, which allows us to use it in a constraint context. A program with placeholders is similar to the representation of an AST with holes that is common in structure editors. Although structure editors are primarily concerned with guaranteeing that the program is well-typed with respect to the abstract syntax signature, recent work investigates editors that also maintain other well-formedness properties, such as well-typedness.

The Hazelnut editor provides a language-parametric structure editor that guarantees well-typed ASTs for languages whose type system is defined in a bidirectional style (Omar, Voysey, Hilton, Aldrich, et al., 2017). JastAdd extends their reference attribute grammars to provide a context-sensitive completion service that suggest the names of variables and functions, but still requires some language-specific effort to derive these suggestions (Söderberg and Hedin, 2011). Steimann et al. use constraint-based language specifications to ensure edits preserve well-formedness (Steimann et al., 2017). They focus on an architecture that allows interaction between the solver and the user during the editing process, to resolve conflicts that may have been introduced. Our aim is to generate semantic completion proposals by combining the mechanisms for syntactic completion, with checking and inference based on the language specification. Another important difference is that issues around name resolution are largely ignored in their work, because references are actual references in the underlying model, whereas in our text-based setting we need to consider naming issues.

Semantic code completion also has similarities to interactive proof search, such as offered by proof assistants. For example, the editor of Agda (Coquand et al., 2006) features holes that are similar to placeholders. An automatic procedure tries to find proof terms (expressions) that fit the goal (type). There are some important differences with our approach to code completion. The procedure to find these terms is not language-parametric, but specific to Agda. The search procedure does not exploit the typing rules, but duplicates knowledge from the type checker. Type correctness is guaranteed by type checking the fragment after it is generated.

Refactoring There is a long line of research on the specification and implementation of refactorings. Tip et al. (2011) study type related refactorings, such as adding type parameters, extracting interfaces, and pulling up methods. They use type constraints to specify the invariants that ensure correct behavior. Steimann and others (Steimann, 2018) extend this work to include constraints for other aspects such as access modifiers and names. By representing the program itself using constraint variables, both the invariants and the refactoring intent can be represented as constraints. Finding the refactored program, within the limits of the given constraints, is delegated to the constraint solver. This approach is in many aspects similar to ours, and hopefully techniques they developed for performance carry over to our approach. An important difference is that by using Statix, the constraint solver is aware of the complete binding model. In their approach preventing capture requires the introduction of inequality constraints between names. These constraints do not follow from regular constraint-based typing rules. In our approach, the resolution constraints that are part of the typing rules can also be used to ensure the invariance of name resolution during refactoring.

6.8 Conclusion

In this paper we have discussed various semantic editor services, and shown how they can be expressed in terms of the semantic rules, constraints, and scope graph. We show that Statix constraints are expressive enough to formulate interesting editor services. We have pointed out that the Statix solver used for type checking is not suitable for the scenarios that arise in editor services, and we have identified several requirements that such alternative solver strategies should have. The main requirements we identified are:

- The solver must be able to try different alternatives, guess values, back-track on failed attempts, and able to return multiple solutions. As we have shown, this is a requirement to implementing code completion, but also for other editor services such as find usages and find all subclasses.

- Instead of always searching for complete solutions (i.e., assignments to all variables), the search should be controlled by user-defined criteria, including whether the solver should only consider deterministic inference, or whether it tries to find solutions non-deterministically. These criteria should be able to depend on variables appearing in the constraints. Specifying which constraint variables may remain unconstrained, and how deep the solver should search for an assignment, allows us to direct the search to finding solutions to only those constraint variables we are interested in, and prevent the solver from getting stuck.

Finally, we propose to extend the mechanisms of creating scopes in Statix to a general mechanism of rigid variables. These rigid variables can be used to solve problems around inventing new concrete names in the constraint solver, and should make it easier to implement various refactorings without having to deal with concrete names, accidental variable capture, and ambiguous names. Factoring out the choice of finding concrete names separates concerns, and also allows for language-specific strategies (e.g., suggesting variables names based on types).

Future Work This paper presented ideas on what would be needed for language-parametric semantic editor services. To verify our approach, we need to implement the proposed extensions to the Statix solver. This will allows us to evaluate their feasibility in practice, as it is not clear whether implementing some of these techniques, such as having the solver back-track and trying to find multiple possible solutions, would cause performance issues or introduce non-termination. And if so, how we could avoid that without impacting the expressiveness of the semantic rules too much.

There are editor services, other than those we discussed, to which we could apply our approach, such as *fix import*, *search for symbol*, and in particular *rename refactoring*. Rename refactoring is interesting because it not only needs to rename the references to the renamed declaration, but possibly other references and declarations as well. For example, when a method is renamed, all overriding methods need to be renamed too, and this relation is not visible in the program model, but only encoded in the semantic rules.

While our current approach is focussed on preserving the static semantics of the program, for certain refactorings it may be required to extend the approach to also preserve certain dynamic aspects of the semantics. Additionally, a combination of our approaches might be used to implement program generation that is guaranteed to produce programs that are semantically correct.

Acknowledgments We thank the anonymous reviewers for their feedback on previous versions of this paper. We also thank Arjen Rouvoet for his comments and his work on Ministatix, an implementation of the core Statix language we use for prototyping. This research was partially funded by the NWO VICI Language Designer's Workbench project (639.023.206).

Conclusion

7

Maurice Ravel, *Concerto pour la main gauche*

This dissertation proposes a meta-language for static semantics, based on scope graphs, that supports directly modeling surface language name binding features, stays close to a familiar inference-style of specification, and allows deriving implementations for type checkers and editor services. In this chapter we reflect on the results of this work, suggest directions for future work, and end with some thoughts on adoption.

7.1 Discussion

The objective of this work, set out in Section 1.4, is to develop a meta-language for the specification of static semantics that is *principled*, *expressive*, *declarative*, *executable*, *reusable*, and *resilient*. In this section we discuss the results according to each of these goals and assess how well we achieved them. When we talk about the implementation, we mean the Statix language as implemented as part of the Spoofax language workbench, unless explicitly stated otherwise.

Principled The meta-language Statix is built on well-defined theories, based on scope graphs as the model for name binding, and on ideas from logic programming, first-order unification, and constraint solving. The core of the meta-language consists of logic predicates, scope graph assertions and queries, and term matching and unification. The semantics of the meta-language, as well as the updates to the scope graph formalisms, are given clear formal descriptions.

We have not conducted user studies to find out if this approach helps users to understand and write specifications. However, informal feedback we have received over the years suggests that users find the language clear and understandable. Most often the difficulty for users is to understand how to apply scope graph concepts to their language.

There are two points of critique here. The first is that the scope graph theory allows references to have multiple interpretations, depending on the context in which they are resolved (the anomalies of Néron et al., 2015). This runs counter to common name binding behavior and leads to confusing and unexpected behavior, indicating a mismatch with the intended domain of the theory. The issue was sidestepped by omitting imports from scope graphs, relying on encodings in the meta-language itself. Unfortunately this means that purely name-based imports are not clearly reflected in the resulting scope graph models anymore.

The second is the complexity of the query construct, which takes several parameters, some of which can be arbitrary user-defined predicates. This was partly the result of attempting to be very general, and partly guided by how the resolution *algorithm* was defined. The result is a monolithic construct that is difficult to understand. The formalization in Chapter 4, which breaks up this monolithic construct, results in a more principled design with simpler, and easier to understand primitives.

Expressive Case studies conducted as part of the work in this dissertation show that a wide range of name binding and type system features can be expressed, including sequential and recursive binding, named imports and exports, absolute and relative qualified names, type-dependent names, nominal and structural subtyping, polymorphic types and generics, and syntactically ambiguous references. At the same time, we can identify some major limitations.

The first major limitation is that inference is restriced to first-order unification. Inferring scope graph structure is not supported, inhibiting, for example, inference of structural record types. An important challenge here is that many scope graphs could support the same usage patterns. Similarly, polymorphic type inference in the style of Hindley-Milner, which requires generalization over free constraint variables, is not supported. This is challenging for two reasons. The first is that types can flow from references to declarations through unification, which means that deciding the free unification variables requires deciding if unresolved queries can resolve to any declaration that is being generalized over. The second is that it is unclear whether generalization should include parts of the scope graph, and how to delimit that.

The second limitation is that predicate rules are required to be syntax directed. It is for example impossible to write a general subsumption rule. It is not obvious what it would take to relax that requirement. For example, practical use of Prolog, which does not have such a restriction, relies heavily on explicit evaluation order and the cut operator, whereas our meta-language design abstracts over all evaluation order.

The third limitation is that it is impossible to express substructural types. These kind of type systems require reasoning about the count and order of definition accesses, for which there are currently no primitives available.

The fourth limitation is that in practice, disambiguation in scope graphs is restricted to locally decidable label ordering. The theory here is more flexible than the implemented resolution algorithm. Chapter 4 shows a design that is more flexible in that regard and allows disambiguation on the full resolution path.

Declarative The meaning of specifications is given by a declarative semantics, which means specifications can be understood without considering operational semantics. Furthermore, the meta-language abstracts over the order of computation, which is pushed completely into the operational semantics. However, as we will discuss below, limitations of the operational semantics sometimes require tweaking specifications to ensure they are executable.

The abstractions that are offered allow effective expression of binding structure and disambiguation policies. The bookkeeping that is often associated with rich environments necessary for non-lexical name binding are reduced to scope passing, which looks very similar to regular lexical environment passing. Similarly, many common typing disciplines and associated type-dependent names are straight-forward to express. User-defined predicates allow introducing auxiliary abstractions, based on the primitives provided by the meta-language, which raises the level of a specification by naming patterns specific to the language.

Good abstractions are missing for the following common language features, resulting in complex encodings. The first is syntactic ambiguity for references. An example is Java, where definitions are organized into separate namespaces, but some references are ambiguous as to which namespace they refer to. This ambiguity cannot be encoded in the scope graph itself, and requires rather complex predicates to encode the decision logic. The second is path-dependent properties. This is clear in the encoding of the substitutions for the FGJ generics, which requires a lot of auxiliary code to build the substitutions that are necessary for type application. The third are name-based operations such as substition and alpha-equivalence. Applying substitutions is a common need in type systems, especially when polymorphic types are involved. Unification in the language obviates the need for explicit substitutions in some cases, in other cases one has to manually write an implementation as part of the specification. Dependently-typed languages are particularly challenging because type equivalence involves type-level computation and alpha-equivalence, which are challenging to specify (Brouwer, 2023).

Executable The work presented in this dissertation focuses on developing an operational semantics and algorithm that allows specifications to be executed as type checkers. It is formally described by an operational semantics and proven correct with respect to the declarative semantics. It is implemented and used for various case studies and as well as in teaching a language engineering course. Later work developed an operational semantics that supports code completion (Daniël A. A. Pelsmaeker et al., 2022).

The main limitation of the operational semantics is that it is incomplete with respect to the declarative semantics. This has generally not caused problems for expressing the specifications for the case studies we conducted, although Rust imports are a known example that runs into this problem. Despite not being a problem for expressiveness, there have been a few occasions where specifications needed to be rewritten slightly to avoid incompleteness.

Implementation performance has been acceptable for smaller projects, but challenging for large projects, although this improved with recent work (Zwaan, 2022; Zwaan et al., 2022). The main issues regarding performance are the following. First, the scope graph resolution algorithm does not allow caching of environments, resulting in duplicate work. The problem is that the environment of a scope is parameterized by the reference that is resolved in that environment, so it cannot be reused when resolving a different reference in the same scope. Second, the interleaving of scope graph construction and query resolution adds a lot of complexity to the implementation. It requires a lot of bookkeeping, as well as speculative query resolution, work that needs to be redone if incomplete scopes are encountered. It also makes the operational semantics and implementation hard to understand. Third, the choice to allow arbitrary predicates as query parameters, instead of restricting queries to simple identifier equality and simple label orders, adds complexity and hurts performance. Evaluating these predicates introduces overhead and inhibits effective optimizations. The formalization in Chapter 4 simplified the query construct significantly, leaving several parts of the query as post-processing steps on the query result set. Initial doubts that this would make incompleteness worse in practice have not materialized, which suggests that this design might be a better trade-off between performance and complexity.

Reusable Both the theory and the implementations have maintained a clear separation between language-specific and language-independent parts, where the latter are reusable for every specification. Reuse within specifications is supported by defining auxiliary predicates as well as support for organizing specifications into separate modules. (Ironically, the surface language of the meta-language itself has not yet been formalized.) Reuse between specifications is limited and shared libraries with reusable language concepts have not been developed.

Resilient The approach to resilience has mostly been a practical one. The constraint solver tries to make as much progress as possible, and turns constraints that remained unsolved into error messages at the end. This approach is quite rudimentary and suffers from typical problems associated with constraint-based type checkers. For example, there is little control over whether error messages appear on the reference or the definition when the expected type at the reference disagrees with the definition type. An unsatisfiable constraint easily leads to cascading errors when other constraints become unsolvable because of unresolved unification variables. A theoretical treatment of errors and partial solutions has not been developed either.

Overall we conclude that our approach is sufficient to express and interpret common name binding and type system features, and that it scales to the full surface syntax of real-world programming languages. The design of Statix meets the goals of principled, expressive, declarative, executable, and reusable design well, although we identified limitations and possibilities for improvement. However, the goal of resilience is only minimally fulfilled and important challenges remain for future research.

7.2 Suggestions for Future Work

The previous section discusses the results of the work in this dissertation. This section gives suggestions for future research to either address some of the limitations we observed, or to take the work in new directions.

Scope Graphs The main theoretical challenge is addressing the unstable interpretation of import references. This requires adapting the scope graph calculus to disallow resolving import references through their importing scopes. This would result in a theory that better fits the static semantics domain. Follow up research could try to develop a limited, more controlled fixed point resolution strategy, perhaps restricted to certain scopes in the graph, that does not suffer from unstable imports.

The developments to increase expressiveness have introduced unnecessary complexities in the theory, particularly in the structure of queries and the shape of data associated with scopes. An important simplification to reduce complexity is to disallow partial matching of scope data, thus simplifying the current predicate query parameters to straight-forward equality checks. The work in Chapter 4 moves in this direction already, which suggests this is feasible without loss of expressiveness or executability.

Reducing complexity opens up possibilities for improving the performance of the resolution algorithm by replacing the top-down search that is currently performed for every reference with a bottom-up computation of environments per scope. Fixing unstable import references makes environments independent of the resolution context. Reducing data matching to equality makes environments independent of query parameters and external context they may capture. These together result in cachable environments and a reduction in duplicate graph traversals.

There are also interesting topics related to increasing expressiveness. The first is supporting disambiguation policies that cannot be decided using a point-wise label order alone. This requires factoring disambiguation out of the current resolution algorithm, as well as developing high-level abstractions to express partial orders over full paths. The second is supporting more complex ways of combining scopes. Currently, scopes are always combined using a union operation. But many languages support renaming imported identifiers, or importing all but a few explicitly excluded identifiers. The challenge is to figure out what combination mechanisms are necessary, how to specify these as part of scopes and edges, and adapt resolution algorithms accordingly. The third is supporting implicitly defining names that are referenced but not defined. An example are logic languages that implicitly define variables scoped in predicate bodies for references that do not appear in the predicate head. This touches on many aspects of the calculus, especially if the inferred declarations are to be added to the graph as new nodes. The fourth is supporting custom path-dependent properties that can be computed as part of resolution. An example is a path-dependent substitution, which is composed from individual substitutions attached to scopes and definitions along a resolution path.

A final avenue for research is to unify ideas from scope graphs with those from stack graphs (Creager and van Antwerpen, 2023). Stack graphs were developed from scope graphs specifically to support incremental resolution. While similar in many aspects, there are several differences between the two formalisms. It would be interesting to see if these differences are mostly cosmetic or more fundamental, and whether the two could be unified into a single theory.

Meta-language Developing new abstractions to increase the expressiveness or the level of specification is an important area of future research. Based on our own experience and the case studies, we suggest the following as valuable additions to the meta-language. The first is syntactic disambiguation through higher-level disambiguation constructs that allow ordered choice and limited search. The second is support for inference of polymorphic types. Determining which unification variables are free to generalize is complicated by the fact that currently declaration types may be inferred from use sites. The last is support for name-based operations such as substitutions and alpha-equivalence, which are important to effectively support polymorphic and dependent types. There is a wealth of research on capture-avoiding and hygienic substitutions that could be used as a starting point.

Major challenges remain around the performance of the meta-language. Both the work in Chapter 5 and later work (Zwaan, 2022; Zwaan et al., 2022) have made gains in runtime performance. But none of these have fundamentally changed the structure of resolution with its drawbacks, such as expensive predicate evaluation in scope graph queries, and fully dynamic constraint scheduling. An interesting experiment, suggested by several people over the years, would be to see if bottom-up evaluation of constraints, similar to the work of Erdweg, Bracevac, et al. (2015), is feasible. Going further would be to eliminate the interleaving of graph construction and querying altogether, which is one of the most complex parts of the operational semantics.

An aspect that has received little attention is the output of executing a specification. Currently the output is ad-hoc and has not always been easy to use for interpretation or program transformations. A principled program representation with clearly defined query and transformation operations would fill this gap and increase the usability for the rest of the compiler pipeline. A starting point would be the work on intrinsically typed representations using scope graphs (Bach Poulsen et al., 2018). Similarly, when a program does not confirm to a specification, the error messages that are produced are not very good. There is existing work on error reporting for constraint-based typing that might be applicable here (e.g., Loncaric et al., 2016; Pavlinovic et al., 2014; D. Zhang et al., 2017; Y. Zhang and Ma, 2014).

There are also possibilities to improve the usability of the meta-language. Specifications can be hard to debug, especially if they trigger incompleteness. Better error messages, clearly visualizing all constraints that contributed to a failed or stuck constraint would be very valuable. There are also opportunities for stronger static checking of specifications. Schema definitions for scope graphs can be used to check that the graph constructed by a specification is correct, and verify that query regular expressions match the graph structure. We have also considered the possibility of statically checking that a specification does not trigger the operational semantics' incompleteness, but so far no viable solutions have been found. Finally, usability would improve with support for reusable predicates and well-defined modules. The ability to import sets of predicates would allow the development of libraries of common language patterns. Developing support for generic and possibly higher-order predicates in the meta-language would greatly benefit such libraries.

Finally, an interesting possible use of specifications is for automated test program generation (or fuzzing). At the moment most fuzzers are hand-written for a specific language or language family. Work by Hatch et al. (2023) uses scope graphs for implementing fuzzers, but does not derive them from existing specifications. Our own experiments in this area have led to approaches for semantic editor services (Daniël A. A. Pelsmaeker et al., 2022), but a general solution for generating whole programs is still missing.

7.3 On Research Adoption

The research in this dissertation always had a strong practical component. Ultimately, the ideas we develop and the tools we implement are meant to improve the work of programmers and language engineers. Ideas spread in the research community through publishing papers, speaking at conferences, or participating in research exchanges. The practice of citations makes it relatively easy to gauge whether the work is being noticed. Making an implementation available can be an even more effective way to spread our research. It can greatly reduce the effort necessary to try and apply our idea, both within and outside academia. Therefore I want to end this dissertation by reflecting on the practical approach of the research, as well as its adoption beyond academia.

The research in this dissertation was conducted in the context of the Spoofax language workbench. The goal of Spoofax is to be a one-stop shop for language engineering, offering an integrated experience, where different aspects of a language are easily developed together (Kats and Visser, 2010). The result is a product that integrates various meta-languages, which is used both at university and in collaboration with industry partners. In a sense it has achieved its goal well. However, the fact that Spoofax has only been used *in collaboration* with industry suggests that adopting it independently is difficult. Let me offer three aspects in which I think the current form of the project hinders independent adoption.

The first is the insistence on full integration. This has resulted in a system that is tightly coupled and where it is difficult to use individual aspects of it without requiring big parts of the system. This makes it more difficult to integrate it in another context (e.g., using Statix to check ASTs coming out of a preexisting parser, or integrating it as a part of a current build pipeline), and using part of the system typically still means introducing many new dependencies. I experienced this myself when trying to set up stand-alone benchmarks for the work in Chapter 5.

The second is weak coherence between the meta-languages. Meta-languages each have their own syntactic choices, and (sometimes subtly) different semantics. Part of this is the result of making languages fit their application domain, and the learning curve for a meta-language is as much about understanding how to apply the underlying concepts as it is about getting the syntax right. But unnecessary divergence makes it more difficult to move from one meta-language aspect to the next. To give an example, Spoofax supports three meta-languages that are based on rules and pattern matching (for program transformation, static semantics, and dynamic semantics). Yet these languages have different rule syntax, slightly different pattern-matching semantics, and quite different module systems.

The third is a focus on designing meta-languages that are expressive and declarative, over good performance. I understand this tendency well, given our focus on solving problems through linguistic abstractions (Visser, 2015). However, performance is important in industry. If performance is not an explicit goal from the start, it can be hard to achieve afterwards, especially for very expressive languages.

An interesting comparison can be made here with stack graphs, an evolution of scope graphs developed at GitHub (Creager and van Antwerpen, 2023). Performance was one of the main reasons why using the original formalism was not feasible, and only after adapting it to allow for incremental resolution could it be used in practice. This does not mean that different priorities would have led to something like stack graphs right away; one idea grows from another. But the improved performance combined with other engineering choices have led to more interest and actual experiments than scope graphs have seen. The first choice is that stack graphs are developed as a library, engineered for stand-alone use, and written in Rust. Besides the fact that Rust helped for performance, it also means the library has a C-compatible interface, which, for better or worse, is still the gold standard for language interoperability. This makes integration in Go, or developing Python bindings, possible, and generally enables integration in many different contexts. The second choice is to provide integration with the Tree-sitter parser framework. Tree-sitter has a big and active ecosystem with grammar definitions for many languages. By connecting to something that is widely available and used already, the barrier of entry is lower. (The fact that GitHub is well known, and the work was presented at *developer* conferences, surely helped, but that alone is not enough to make people actually use something.)

Of course the comparison is to some degree between apples and oranges. Academia and industry have different goals and requirements. The relative independence of academic researchers and short-lived nature of PhD projects, combined with an insistence on novelty, makes long-term refinement and practical engineering difficult. But the past success of the Spoofax project shows that it is possible to develop a research culture that does result in long-lived, practical software that goes beyond throw-away prototypes. And some of that work is already happening, for example by breaking up the monolithic build pipeline (Konat et al., 2018), or developing standalone frameworks for scope graph based type checkers (Bach Poulsen et al., 2023). I believe this research culture can evolve and lead to a broader adoption of the work we do.

Bibliography

Abadi, Martín, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy (1991). "Explicit Substitutions." In: *Journal of Functional Programming* 1.4, pp. 375–416 (cit. on p. 58).

Agha, Gul A. (1990). *ACTORS - a model of concurrent computation in distributed systems*. MIT Press series in artificial intelligence. MIT Press. ISBN: 978-0-262-01092-4 (cit. on p. 140).

Alblas, Henk (1991). "Attribute Evaluation Methods." In: *Attribute Grammars, Applications and Systems, International Summer School SAGA, Prague, Czechoslovakia, June 4-13, 1991, Proceedings*. Ed. by Henk Alblas and Borivoj Melichar. Vol. 545. Lecture Notes in Computer Science. Springer, pp. 48–113. ISBN: 3-540-54572-7 (cit. on p. 120).

Amin, Nada, Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki (2016). "The Essence of Dependent Object Types." In: *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*. Ed. by Sam Lindley, Conor McBride, Philip W. Trinder, and Donald Sannella. Vol. 9600. Lecture Notes in Computer Science. Springer, pp. 249–272. DOI: 10.1007/978-3-319-30936-1_14. (Cit. on p. 66).

Amin, Nada and Tiark Rompf (2017). "Type soundness proofs with definitional interpreters." In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. Ed. by Giuseppe Castagna and Andrew D. Gordon. ACM, pp. 666–679. ISBN: 978-1-4503-4660-3. (Cit. on p. 66).

Appel, Andrew W. (1998). *Modern Compiler Implementation in Java*. Cambridge University Press. ISBN: 0-521-58388-8 (cit. on p. 128).

Astarte, Troy Kaighin (2019). "Formalising Meaning: a History of Programming Language Semantics." PhD thesis. Newcastle upon Tyne: Newcastle University (cit. on p. 5).

Baader, Franz and Tobias Nipkow (1998). *Term rewriting and all that*. Cambridge University Press. ISBN: 978-0-521-45520-6 (cit. on p. 42).

Bach Poulsen, Casper, Pierre Néron, Andrew P. Tolmach, and Eelco Visser (2016). "Scopes Describe Frames: A Uniform Model for Memory Layout in Dynamic Semantics." In: *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*. Ed. by Shriram Krishnamurthi and Benjamin S. Lerner. Vol. 56. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik. DOI: 10.4230/LIPIcs.ECOOP.2016.20 (cit. on pp. 48, 68, 81).

Bach Poulsen, Casper, Arjen Rouvoet, Andrew P. Tolmach, Robbert Krebbers, and Eelco Visser (2018). "Intrinsically-typed definitional interpreters for imperative languages." In: *Proceedings of the ACM on Programming Languages* 2.POPL. DOI: 10.1145/3158104. (Cit. on pp. 48, 68, 81, 185).

Bach Poulsen, Casper, Aron Zwaan, and Paul Hübner (2023). "A Monadic Framework for Name Resolution in Multi-phased Type Checkers." In: *Proceedings of the 22nd ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2023, Cascais, Portugal, October 22-23, 2023*. Ed. by Coen De Roover, Bernhard Rumpe, and Amir Shaikhha. ACM, pp. 14–28. DOI: 10.1145/3624007.3624051. (Cit. on p. 187).

Backus, John W., Friedrich L. Bauer, Julien Green, C. Katz, John McCarthy, Alan J. Perlis, Heinz Rutishauser, Klaus Samelson, Bernard Vauquois, Joseph Henry Wegstein, Adriaan van Wijngaarden, and Michael Woodger (1960). "Report on the algorithmic language ALGOL 60." In: *Communications of the ACM* 3.5, pp. 299–314. DOI: 10.1145/367236.367262. (Cit. on p. 4).

Bird, Richard S. (1984). “Using Circular Programs to Eliminate Multiple Traversals of Data.” In: *Acta Informatica* 21, pp. 239–250 (cit. on p. 121).

Boyland, J. (1996). “Descriptional Composition of Compiler Components.” PhD thesis (cit. on p. 121).

Boyland, John Tang (2005). “Remote attribute grammars.” In: *Journal of the ACM* 52.4, pp. 627–687. DOI: 10.1145/1082036.1082042. (Cit. on pp. 97, 121).

Brouwer, Jonathan (2023). “Dependently Typed Languages in Statix.” MA thesis. Delft, The Netherlands: Delft University of Technology. URL: http://resolver.tudelft.nl/uuid:7bf3c0f5-71fb-4e08-bcdb-1c873c7e1e63 (cit. on p. 181).

Brzozowski, Janusz A. (1964). “Derivatives of Regular Expressions.” In: *Journal of the ACM* 11.4, pp. 481–494 (cit. on pp. 38, 100, 146).

Cardelli, Luca (1988). “Structural Subtyping and the Notion of Power Type.” In: *POPL*, pp. 70–79 (cit. on p. 56).

Chandy, K. Mani, Jayadev Misra, and Laura M. Haas (1983). “Distributed Deadlock Detection.” In: *ACM Trans. Comput. Syst.* 1.2, pp. 144–156. DOI: 10.1145/357360.357365. (Cit. on p. 149).

Chang, Stephen, Alex Knauth, and Ben Greenman (2017). “Type systems as macros.” In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. Ed. by Giuseppe Castagna and Andrew D. Gordon. ACM, pp. 694–705. ISBN: 978-1-4503-4660-3. (Cit. on p. 81).

Coquand, Catarina, Makoto Takeyama, and Dan Synek (2006). “An Emacs-Interface for Type-Directed Supportfor Constructing Proofs and Programs.” In: *European Joint Conferences on Theory and Practice of Software, ENTCS*. Vol. 2 (cit. on p. 175).

Creager, Douglas A. and Hendrik van Antwerpen (2023). “Stack Graphs: Name Resolution at Scale.” In: *Eelco Visser Commemorative Symposium (EVCS 2023)*. Ed. by Ralf Lämmel, Peter D. Mosses, and Friedrich Steimann. Vol. 109. Open Access Series in Informatics (OASIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 8:1–8:12. DOI: 10.4230/OASIcs.EVCS.2023.8. (Cit. on pp. 184, 186, 202).

Damas, Luís and Robin Milner (1982). “Principal Type-Schemes for Functional Programs.” In: *POPL*, pp. 207–212 (cit. on p. 82).

De Souza Amorim, Luis Eduardo, Sebastian Erdweg, Guido Wachsmuth, and Eelco Visser (2016). “Principled syntactic code completion using placeholders.” In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering, Amsterdam, The Netherlands, October 31 - November 1, 2016*. Ed. by Tijs van der Storm, Emilie Balland, and Dániel Varró. ACM, pp. 163–175. DOI: 10.1145/2997364.2997374. (Cit. on pp. 169, 175).

ECMA International (2017). *Standard ECMA-334: C# Language Specification, 5th edition*. URL: https://www.ecma-international.org/publications-and-standards/standards/ecma-334/ (cit. on p. 4).

Ekman, Torbjörn and Görel Hedin (2006). “Modular Name Analysis for Java Using JastAdd.” In: *Generative and Transformational Techniques in Software Engineering, International Summer School, GTTSE 2005, Braga, Portugal, July 4-8, 2005. Revised Papers*. Ed. by Ralf Lämmel, João Saraiva, and Joost Visser. Vol. 4143. Lecture Notes in Computer Science. Springer, pp. 422–436. DOI: 10.1007/11877028_18. (Cit. on pp. 45, 81, 82, 121).

— (2007a). “The JastAdd extensible Java compiler.” In: *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*. Ed. by Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr. ACM, pp. 1–18. DOI: 10.1145/1297027.1297029. (Cit. on pp. 45, 81, 121).

— (2007b). “The JastAdd system - modular extensible compiler construction.” In: *Science of Computer Programming* 69.1-3, pp. 14–26. DOI: 10.1016/j.scico.2007.02.003. (Cit. on p. 121).

Ellison III, Charles M. (2008). "A Rewriting Logic Approach to Defining Type Systems." MA thesis. University of Illinois at Urbana-Champaign. URL: http://hdl.handle.net/2142/18078 (cit. on p. 81).

Erdweg, Sebastian, Oliver Bracevac, Edlira Kuci, Matthias Krebs, and Mira Mezini (2015). "A co-contextual formulation of type rules and its application to incremental type checking." In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. Ed. by Jonathan Aldrich and Patrick Eugster. ACM, pp. 880–897. DOI: 10.1145/2814270.2814277. (Cit. on pp. 44, 184).

Erdweg, Sebastian, Moritz Lichter, and Manuel Weiel (2015). "A sound and optimal incremental build system with dynamic dependencies." In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. Ed. by Jonathan Aldrich and Patrick Eugster. ACM, pp. 89–106. DOI: 10.1145/2814270.2814316. (Cit. on p. 155).

Erdweg, Sebastian, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth, and Jimi van der Woning (2013). "The State of the Art in Language Workbenches - Conclusions from the Language Workbench Challenge." In: *Software Language Engineering - 6th International Conference, SLE 2013, Indianapolis, IN, USA, October 26-28, 2013. Proceedings*. Ed. by Martin Erwig, Richard F. Paige, and Eric Van Wyk. Vol. 8225. Lecture Notes in Computer Science. Springer, pp. 197–217. DOI: 10.1007/978-3-319-02654-1_11. (Cit. on pp. 160, 161, 174).

Erdweg, Sebastian, Tijs van der Storm, Markus Völter, Laurence Tratt, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth, and Jimi van der Woning (2015). "Evaluating and comparing language workbenches: Existing results and benchmarks for the future." In: *Computer Languages, Systems & Structures* 44, pp. 24–47. DOI: 10.1016/j.cl.2015.08.007. (Cit. on pp. 7, 17).

Eysholdt, M. and H. Behrens (2010). "Xtext: implement your language faster than the quick and dirty way." In: *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*. ACM, pp. 307–309 (cit. on p. 174).

Felleisen, Matthias, Robby Findler, and Matthew Flatt (2009). *Semantics Engineering with PLT Redex*. MIT Press. ISBN: 978-0-262-06275-6. (Cit. on p. 81).

Fetscher, Burke, Koen Claessen, Michal H. Palka, John Hughes, and Robby Findler (2015). "Making Random Judgments: Automatically Generating Well-Typed Terms from the Definition of a Type-System." In: *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*. Ed. by Jan Vitek. Vol. 9032. Lecture Notes in Computer Science. Springer, pp. 383–405. DOI: 10.1007/978-3-662-46669-8_16. (Cit. on p. 81).

Flatt, Matthew (2016). "Binding as sets of scopes." In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. Ed. by Rastislav Bodik and Rupak Majumdar. ACM, pp. 705–717. DOI: 10.1145/2837614.2837620. (Cit. on p. 81).

Fowler, Martin (2005). *Language Workbenches: The Killer-App for Domain Specific Languages?* URL: http://www.martinfowler.com/articles/languageWorkbench.html (cit. on pp. 7, 160).

Frühwirth, Thom (2009). *Constraint Handling Rules*. Cambridge University Press. ISBN: 978-0-521-87776-3 (cit. on p. 82).

Frühwirth, Thom and Pascal Brisset (1995). *High-Level Implementations of Constraint Handling Rules*. Technical Report ECRC-TR-95-20 (cit. on p. 74).

Frühwirth, Thom W. (1998). “Theory and Practice of Constraint Handling Rules.” In: *Journal of Logic and Algebraic Programming* 37.1-3, pp. 95–138 (cit. on p. 119).

GCC (n.d.). *The Parallel GCC*. URL: https://gcc.gnu.org/wiki/ParallelGcc (cit. on p. 125).

Girard, Jean-Yves (1972). “Interprétation fonctionnelle et élimination des coupures de l’arithmétique d’ordre supérieur.” French. PhD thesis. Université Paris 7 (cit. on pp. 49, 50).

Go (n.d.). *Go 1.9: Parallel Compilation*. URL: https://golang.org/doc/go1.9#parallel-compile (cit. on p. 154).

Gosling, James, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, and Daniel Smith (2018). *The Java Language Specification, Java SE 11 Edition* (cit. on p. 4).

Hatch, William Gallard, Pierce Darragh, Sorawee Porncharoenwase, Guy Watson, and Eric Eide (2023). “Generating Conforming Programs with Xsmith.” In: *Proceedings of the 22nd ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2023, Cascais, Portugal, October 22-23, 2023*. Ed. by Coen De Roover, Bernhard Rumpe, and Amir Shaikhha. ACM, pp. 86–99. DOI: 10.1145/3624007.3624056. (Cit. on p. 185).

Hedin, Görel (2000). “Reference Attributed Grammars.” In: *Informatica (Slovenia)* 24.3, pp. 301–317 (cit. on pp. 45, 93, 121).

— (2009). “An Introductory Tutorial on JastAdd Attribute Grammars.” In: *Generative and Transformational Techniques in Software Engineering III - International Summer School, GTTSE 2009, Braga, Portugal, July 6-11, 2009. Revised Papers*. Ed. by Joao M. Fernandes, Ralf Lämmel, Joost Visser, and João Saraiva. Vol. 6491. Lecture Notes in Computer Science. Springer, pp. 166–200. DOI: 10.1007/978-3-642-18023-1_4. (Cit. on p. 81).

Heeren, Bastiaan, Jurriaan Hage, and S. Doaitse Swierstra (2003). “Scripting the type inference process.” In: *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming, ICFP 2003, Uppsala, Sweden, August 25-29, 2003*. Ed. by Colin Runciman and Olin Shivers. ACM, pp. 3–13. DOI: 10.1145/944705.944707. (Cit. on p. 46).

Helm, Dominik, Florian Kübler, Jan Thomas Kölzer, Philipp Haller, Michael Eichberg, Guido Salvaneschi, and Mira Mezini (2020). “A programming model for semi-implicit parallelization of static analyses.” In: *ISSTA ’20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020*. Ed. by Sarfraz Khurshid and Corina S. Pasareanu. ACM, pp. 428–439. DOI: 10.1145/3395363.3397367. (Cit. on p. 156).

Hindley, Roger (1969). “The Principal Type-Scheme of an Object in Combinatory Logic.” In: *Trans. Amer. Math. Soc* 146. DOI: 10.2307/1995158 (cit. on p. 82).

Igarashi, Atsushi, Benjamin C. Pierce, and Philip Wadler (2001). “Featherweight Java: a minimal core calculus for Java and GJ.” In: *ACM Transactions on Programming Languages and Systems* 23.3, pp. 396–450. DOI: 10.1145/503502.503505. (Cit. on pp. 49, 50, 60, 63, 64, 81).

Johnsson, Thomas (1987). “Attribute grammars as a functional programming paradigm.” In: *Functional Programming Languages and Computer Architecture, Portland, Oregon, USA, September 14-16, 1987, Proceedings*. Ed. by Gilles Kahn. Vol. 274. Lecture Notes in Computer Science. Springer, pp. 154–173. DOI: 10.1007/3-540-18317-5_10. (Cit. on p. 121).

Kastens, Uwe and William M. Waite (1991). “An Abstract Data Type for Name Analysis.” In: *Acta Informatica* 28.6, pp. 539–558. DOI: 10.1007/BF01463944 (cit. on p. 45).

Kats, Lennart C. L., Rob Vermaas, and Eelco Visser (2011). “Testing domain-specific languages.” In: *Companion to the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*. Ed. by Cristina Videira Lopes and Kathleen Fisher. ACM, pp. 25–26. DOI: 10.1145/2048147.2048160. (Cit. on p. 78).

Kats, Lennart C. L. and Eelco Visser (2010). “The Spoofax language workbench: rules for declarative specification of languages and IDEs.” In: *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-*

Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010. Ed. by William R. Cook, Siobhán Clarke, and Martin C. Rinard. Reno/Tahoe, Nevada: ACM, pp. 444–463. DOI: 10.1145/1869459.1869497. (Cit. on pp. 7, 9, 13, 43, 44, 78, 156, 174, 186).

Keuchel, Steven, Stephanie Weirich, and Tom Schrijvers (2016). "Needle & Knot: Binder Boilerplate Tied Up." In: *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*. Ed. by Peter Thiemann. Vol. 9632. Lecture Notes in Computer Science. Springer, pp. 419–445. DOI: 10.1007/978-3-662-49498-1_17. (Cit. on p. 80).

Knuth, Donald E. (1968). "Semantics of Context-Free Languages." In: *Theory Comput. Syst.* 2.2, pp. 127–145. DOI: 10.1007/BF01692511. (Cit. on pp. 6, 45).

Knuth, Donald E. and Luis Trabb Pardo (1980). "The Early Development of Programming Languages." In: *A History of Computing in the Twentieth Century*. San Diego: Academic Press, pp. 197–273. DOI: 10.1016/B978-0-12-491650-0.50019-8. (Cit. on p. 1).

Konat, Gabriël, Sebastian Erdweg, and Eelco Visser (2018). "Scalable incremental building with dynamic task dependencies." In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. Ed. by Marianne Huchard, Christian Kästner, and Gordon Fraser. ACM, pp. 76–86. DOI: 10.1145/3238147.3238196. (Cit. on pp. 155, 187).

Konat, Gabriël, Lennart C. L. Kats, Guido Wachsmuth, and Eelco Visser (2012). "Declarative Name Binding and Scope Rules." In: *Software Language Engineering, 5th International Conference, SLE 2012, Dresden, Germany, September 26-28, 2012, Revised Selected Papers*. Ed. by Krzysztof Czarnecki and Görel Hedin. Vol. 7745. Lecture Notes in Computer Science. Springer, pp. 311–331. DOI: 10.1007/978-3-642-36089-3_18. (Cit. on pp. 19, 43, 156, 175).

Kulkarni, Milind, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew (2007). "Optimistic parallelism requires abstractions." In: *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*. Ed. by Jeanne Ferrante and Kathryn S. McKinley. ACM, pp. 211–222. DOI: 10.1145/1250734.1250759. (Cit. on p. 156).

Kuper, Lindsey, Aaron Turon, Neelakantan R. Krishnaswami, and Ryan R. Newton (2014). "Freeze after writing: quasi-deterministic parallel programming with LVars." In: *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*. Ed. by Suresh Jagannathan and Peter Sewell. ACM, pp. 257–270. DOI: 10.1145/2535838.2535842. (Cit. on p. 155).

Lampropoulos, Leonidas, Diane Gallois-Wong, Catalin Hritcu, John Hughes, Benjamin C. Pierce, and Li-yao Xia (2017). "Beginner's luck: a language for property-based generators." In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. Ed. by Giuseppe Castagna and Andrew D. Gordon. ACM, pp. 114–129. DOI: 10.1145/3009837.3009868. (Cit. on p. 81).

Loncaric, Calvin, Satish Chandra 0001, Cole Schlesinger, and Manu Sridharan (2016). "A practical framework for type inference error explanation." In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*. Ed. by Eelco Visser and Yannis Smaragdakis. ACM, pp. 781–799. DOI: 10.1145/2983990.2983994. (Cit. on p. 185).

Magnusson, Eva, Torbjörn Ekman, and Görel Hedin (2009). "Demand-driven evaluation of collection attributes." In: *Automated Software Engineering* 16.2, pp. 291–322. DOI: 10.1007/s10515-009-0046-z. (Cit. on p. 122).

Magnusson, Eva and Görel Hedin (2003). "Circular Reference Attributed Grammars - Their Evaluation and Applications." In: *Electronic Notes in Theoretical Computer Science* 82.3, pp. 532–554. DOI: 10.1016/S1571-0661(05)82627-1. (Cit. on p. 123).

Matthews, David C. J. and Makarius Wenzel (2010). “Efficient parallel programming in Poly/ML and Isabelle/ML.” In: *Proceedings of the POPL 2010 Workshop on Declarative Aspects of Multicore Programming, DAMP 2010, Madrid, Spain, January 19, 2010*. Ed. by Leaf Petersen and Enrico Pontelli. ACM, pp. 53–62. DOI: 10.1145/1708046.1708058. (Cit. on p. 155).

Mens, Tom, Serge Demeyer, Bart Du Bois, Hans Stenten, and Pieter Van Gorp (2003). “Refactoring: Current Research and Future Trends.” In: *Electronic Notes in Theoretical Computer Science* 82.3, pp. 483–499. DOI: 10.1016/S1571-0661(05)82624-6. (Cit. on p. 159).

Mensing, Adrian D., Hendrik van Antwerpen, Casper Bach Poulsen, and Eelco Visser (2019). “From definitional interpreter to symbolic executor.” In: *Proceedings of the 4th ACM SIGPLAN International Workshop on Meta-Programming Techniques and Reflection*. META 2019. Athens, Greece: Association for Computing Machinery, pp. 11–20. DOI: 10.1145/3358502.3361269 (cit. on p. 202).

Milner, Robin (1978). “A Theory of Type Polymorphism in Programming.” In: *J. Comput. Syst. Sci.* 17.3, pp. 348–375. DOI: 10.1016/0022-0000(78)90014-4 (cit. on pp. 44, 82).

Milner, Robin, Mads Tofte, Robert Harper, and David MacQueen (1997). *The Definition of Standard ML, Revised*. Cambridge, MA, USA: MIT Press. ISBN: 978-0-262-28700-5 (cit. on pp. 5, 6, 93).

Moss, J. Eliot B. (1986). “Getting the Operating System Out of the Way.” In: *IEEE Data Eng. Bull.* 9.3, pp. 35–42. URL: http://sites.computer.org/debull/86SEP-CD.pdf (cit. on p. 119).

Mulligan, Dominic P., Scott Owens, Kathryn E. Gray, Tom Ridge, and Peter Sewell (2014). “Lem: reusable engineering of real-world semantics.” In: *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*. Ed. by Johan Jeuring and Manuel M. T. Chakravarty. ACM, pp. 175–188. DOI: 10.1145/2628136.2628143. (Cit. on p. 80).

Murphy, Gail C., Mik Kersten, and Leah Findlater (2006). “How Are Java Software Developers Using the Eclipse IDE?” In: *IEEE Software* 23.4, pp. 76–83. DOI: 10.1109/MS.2006.105. (Cit. on p. 159).

Néron, Pierre, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth (2015). “A Theory of Name Resolution.” In: *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*. Ed. by Jan Vitek. Vol. 9032. Lecture Notes in Computer Science. Springer, pp. 205–231. DOI: 10.1007/978-3-662-46669-8_9. (Cit. on pp. xi, xiii, 7, 11, 17–20, 33, 36, 43, 44, 48, 50, 51, 68, 90, 95, 100, 120, 126, 129, 156, 163, 179).

Newton, Ryan R., Ömer S. Agacan, Peter P. Fogg, and Sam Tobin-Hochstadt (2016). “Parallel type-checking with haskell using saturating LVars and stream generators.” In: *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2016, Barcelona, Spain, March 12-16, 2016*. Ed. by Rafael Asenjo 0001 and Tim Harris. ACM, p. 6. DOI: 10.1145/2851141.2851142. (Cit. on p. 155).

O’Hearn, Peter W., John C. Reynolds, and Hongseok Yang (2001). “Local Reasoning about Programs that Alter Data Structures.” In: *Computer Science Logic, 15th International Workshop, CSL 2001. 10th Annual Conference of the EACSL, Paris, France, September 10-13, 2001, Proceedings*. Ed. by Laurent Fribourg. Vol. 2142. Lecture Notes in Computer Science. Springer, pp. 1–19. DOI: 10.1007/3-540-44802-0_1. (Cit. on p. 101).

Odersky, Martin, Philippe Altherr, Vincent Cremet, Gilles Dubochet, Burak Emir, Philipp Haller, Stéphane Micheloud, Nikolay Mihaylov, Adriaan Moors, Lukas Rytz, Michel Schinz, Erik Stenman, and Matthias Zenger (2019). *Scala Language Specification: Version 2.13*. URL: https://www.scala-lang.org/files/archive/spec/2.13/ (cit. on p. 4).

Odersky, Martin, Martin Sulzmann, and Martin Wehr (1999). “Type Inference with Constrained Types.” In: *TAPOS* 5.1, pp. 35–55 (cit. on pp. 82, 120).

Odersky, Martin, Christoph Zenger, and Matthias Zenger (2001). “Colored local type inference.” In: *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL. Association for Computing Machinery, pp. 41–53. DOI: 10.1145/360204.360207. (Cit. on p. 82).

Omar, Cyrus, Ian Voysey, Michael Hilton, Jonathan Aldrich, and Matthew A. Hammer (2017). "Hazelnut: a bidirectionally typed structure editor calculus." In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. Ed. by Giuseppe Castagna and Andrew D. Gordon. ACM, pp. 86–99. DOI: 10.1145/3009837.3009900. (Cit. on p. 175).

Omar, Cyrus, Ian Voysey, Michael Hilton, Joshua Sunshine, Claire Le Goues, Jonathan Aldrich, and Matthew A. Hammer (2017). "Toward Semantic Foundations for Program Editors." In: *2nd Summit on Advances in Programming Languages, SNAPL 2017, May 7-10, 2017, Asilomar, CA, USA*. Ed. by Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi. Vol. 71. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik. DOI: 10.4230/LIPIcs.SNAPL.2017.11. (Cit. on pp. 160, 174).

OpenJDK (n.d.). *Java Microbenchmark Harness (JMH)*. URL: https://openjdk.java.net/projects/code-tools/jmh/ (cit. on p. 151).

Öqvist, Jesper and Görel Hedin (2017). "Concurrent circular reference attribute grammars." In: *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2017, Vancouver, BC, Canada, October 23-24, 2017*. Ed. by Benoît Combemale, Marjan Mernik, and Bernhard Rumpe. ACM, pp. 151–162. DOI: 10.1145/3136014.3136032. (Cit. on p. 155).

Palsberg, Jens and Michael I. Schwartzbach (1991). "Object-Oriented Type Inference." In: *OOPSLA*, pp. 146–161 (cit. on p. 44).

— (1994). *Object-oriented type systems*. Wiley professional computing. Wiley. ISBN: 978-0-471-94128-6 (cit. on p. 44).

Pavlinovic, Zvonimir, Tim King, and Thomas Wies (2014). "Finding minimum type error sources." In: *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*. Ed. by Andrew P. Black and Todd D. Millstein. ACM, pp. 525–542. DOI: 10.1145/2660193.2660230. (Cit. on p. 185).

Pelsmaeker, Daniel A. A., Hendrik van Antwerpen, and Eelco Visser (2019). "Towards Language-Parametric Semantic Editor Services Based on Declarative Type System Specifications (Brave New Idea Paper)." In: *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*. Ed. by Alastair F. Donaldson. Vol. 134. LIPIcs. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik. DOI: 10.4230/LIPIcs.ECOOP.2019.26 (cit. on pp. 12, 122, 159, 201).

Pelsmaeker, Daniël A. A., Hendrik van Antwerpen, Casper Bach Poulsen, and Eelco Visser (2022). "Language-parametric static semantic code completion." In: *Proceedings of the ACM on Programming Languages* 6.OOPSLA, pp. 1–30. DOI: 10.1145/3527329. (Cit. on pp. 181, 185, 202).

Pierce, Benjamin C. (2002). *Types and Programming Languages*. Cambridge, Massachusetts: MIT Press (cit. on pp. 47, 49, 50, 56, 58, 60, 68).

Pierce, Benjamin C. and David N. Turner (2000). "Local type inference." In: *ACM Transactions on Programming Languages and Systems* 22.1, pp. 1–44. DOI: 10.1145/345099.345100. (Cit. on p. 82).

Pottier, François and Diddier Rémy (2005). "The Essence of ML Type Inference." In: *Advanced Topics in Types and Programming Languages*. Ed. by Benjamin C. Pierce. The MIT Press, pp. 389–489. ISBN: 0-262-16228-8 (cit. on pp. 72, 82, 120).

Reali, Patrik (2000). "Structuring a Compiler with Active Objects." In: *Modular Programming Languages, Joint Modular Languages Conference, JMLC 2000, Zurich, Switzerland, September 6-8, 2000, Proceedings*. Ed. by Jürg Gutknecht and Wolfgang Weck. Vol. 1897. Lecture Notes in Computer Science. Springer, pp. 250–262. DOI: 10.1007/10722581_20 (cit. on p. 154).

Reynolds, John C. (1974). "Towards a theory of type structure." In: *Programming Symposium, Proceedings Colloque sur la Programmation, Paris, France, April 9-11, 1974*. Ed. by Bernard Robinet. Vol. 19. Lecture Notes in Computer Science. Springer, pp. 408–423. DOI: 10.1007/3-540-06859-7_148 (cit. on pp. 49, 50).

Robillard, Martin P., Wesley Coelho, and Gail C. Murphy (2004). "How Effective Developers Investigate Source Code: An Exploratory Study." In: *IEEE Trans. Software Eng.* 30.12, pp. 889–903. DOI: 10.1109/TSE.2004.101. (Cit. on p. 159).

Rodriguez, Jonathan and Ondrej Lhoták (2011). "Actor-Based Parallel Dataflow Analysis." In: *Compiler Construction - 20th International Conference, CC 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*. Ed. by Jens Knoop. Vol. 6601. Lecture Notes in Computer Science. Springer, pp. 179–197. DOI: 10.1007/978-3-642-19861-8_11. (Cit. on p. 156).

Rosu, Grigore and Traian-Florin Serbanuta (2010). "An overview of the K semantic framework." In: *Journal of Logic and Algebraic Programming* 79.6, pp. 397–434. DOI: 10.1016/j.jlap.2010.03.012. (Cit. on p. 81).

Rouvoet, Arjen, Hendrik van Antwerpen, Casper Bach Poulsen, Robbert Krebbers, and Eelco Visser (2020a). "Knowing when to ask: sound scheduling of name resolution in type checkers derived from declarative specifications." In: *Proceedings of the ACM on Programming Languages* 4.OOPSLA. DOI: 10.1145/3428248 (cit. on pp. 12, 87, 126, 127, 129, 131, 132, 140, 151, 153, 156, 202).

Rouvoet, Arjen, Hendrik van Antwerpen, Casper Bach Poulsen, Robbert Krebbers, and Eelco Visser (2020b). *Knowing When to Ask: Artifact*. Version 1.0. DOI: 10.5281/zenodo.4068065. (Cit. on p. 117).

— (2020c). "Knowing When to Ask: Sound scheduling of name resolution in type checkers derived from declarative specifications (Extended Version)." In: Zenodo. DOI: 10.5281/zenodo.4091445. (Cit. on pp. 90, 104, 105, 108, 116, 118).

Rust (n.d.). *Parallel Compilation*. URL: https://rustc-dev-guide.rust-lang.org/parallel-rustc.html (cit. on p. 154).

Sasaki, Akira and Masataka Sassa (2003). "Circular Attribute Grammars with Remote Attribute References and their Evaluators." In: *New Generation Comput.* 22.1, pp. 37–60. DOI: 10.1007/BF03037280. (Cit. on p. 123).

Schäfer, Max and Oege de Moor (2010). "Specifying and implementing refactorings." In: *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010*. Ed. by William R. Cook, Siobhán Clarke, and Martin C. Rinard. Reno/Tahoe, Nevada: ACM, pp. 286–301. DOI: 10.1145/1869459.1869485. (Cit. on p. 160).

Schäfer, Max, Andreas Thies, Friedrich Steimann, and Frank Tip (2012). "A Comprehensive Approach to Naming and Accessibility in Refactoring Java Programs." In: *IEEE Trans. Software Eng.* 38.6, pp. 1233–1257. DOI: 10.1109/TSE.2012.13. (Cit. on p. 160).

Schäfer, Steven, Tobias Tebbi, and Gert Smolka (2015). "Autosubst: Reasoning with de Bruijn Terms and Parallel Substitutions." In: *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings*. Ed. by Christian Urban and Xingyuan Zhang. Vol. 9236. Lecture Notes in Computer Science. Springer, pp. 359–374. DOI: 10.1007/978-3-319-22102-1_24. (Cit. on p. 80).

Seshadri, V., S. Weber, D. B. Wortman, C. P. Yu, and I. Small (1988). "Semantic analysis in a concurrent compiler." In: *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*. PLDI '88. Atlanta, Georgia, USA: Association for Computing Machinery, pp. 233–240. DOI: 10.1145/53990.54013. (Cit. on p. 129).

Sewell, Peter, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strnisa (2010). "Ott: Effective tool support for the working semanticist." In: *Journal of Functional Programming* 20.1, pp. 71–122. DOI: 10.1017/S0956796809990293. (Cit. on p. 80).

Shao, Zhong and Andrew W. Appel (1993). "Smartest Recompilation." In: *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 439–450. DOI: 10.1145/158511.158702 (cit. on p. 126).

Simonet, Vincent and François Pottier (2007). "A constraint-based approach to guarded algebraic data types." In: *ACM Transactions on Programming Languages and Systems* 29.1, p. 1. DOI: 10.1145/1180475.1180476. (Cit. on p. 82).

Söderberg, Emma and Görel Hedin (2011). "Building semantic editors using JastAdd: tool demonstration." In: *Language Descriptions, Tools and Applications, LDTA 2011, Saarbrücken, Germany, March 26-27, 2011. Proceeding*. Ed. by Claus Brabrand and Eric Van Wyk. ACM, p. 11. DOI: 10.1145/1988783.1988794. (Cit. on p. 175).

StackExchange (n.d.[a]). *Do compilers utilize multithreading for faster compile times?* URL: https://softwareengineering.stackexchange.com/questions/322494/do-compilers-utilize-multithreading-for-faster-compile-times (cit. on p. 154).

— (n.d.[b]). *Is there something that prevents a multithreaded C# compiler implementation?* URL: https://softwareengineering.stackexchange.com/questions/330026/is-there-something-that-prevents-a-multithreaded-c-compiler-implementation (cit. on p. 154).

— (n.d.[c]). *Why isn't Javac running on multiple cores?* URL: https://stackoverflow.com/questions/46461757/why-isnt-javac-running-on-multiple-cores (cit. on p. 154).

Stallman, Richard M., Roland McGrath, and Paul D. Smith (2016). *GNU Make*. Free Software Foundation (cit. on p. 155).

Stansifer, Paul (2016). "Flexible binding-safe programming." PhD thesis. Northeastern University. DOI: 10.17760/D20213100 (cit. on p. 81).

Steimann, Friedrich (2018). "Constraint-Based Refactoring." In: *ACM Transactions on Programming Languages and Systems* 40.1. DOI: 10.1145/3156016. (Cit. on pp. 160, 172, 176).

Steimann, Friedrich, Marcus Frenkel, and Markus Voelter (2017). "Robust projectional editing." In: *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2017, Vancouver, BC, Canada, October 23-24, 2017*. Ed. by Benoît Combemale, Marjan Mernik, and Bernhard Rumpe. ACM, pp. 79–90. DOI: 10.1145/3136014.3136034. (Cit. on p. 175).

Strnisa, Rok and Matthew J. Parkinson (2011). "Lightweight Java." In: *Archive of Formal Proofs* 2011. URL: http://afp.sourceforge.net/entries/LightweightJava.shtml (cit. on p. 81).

Sulzmann, Martin and Peter J. Stuckey (2008). "HM(X) type inference is CLP(X) solving." In: *Journal of Functional Programming* 18.2, pp. 251–283. DOI: 10.1017/S0956796807006569. (Cit. on p. 82).

Swift (n.d.). *Swift Compiler Performance*. URL: https://github.com/swiftlang/swift/blob/6221b29c6835442706fbb44b67b755d370a87d96/docs/CompilerPerformance.md (cit. on p. 154).

T.B. Steel, Jr., ed. (1966). *Formal Language Description Languages for Computer Programming*. Baden-bei-Wien: North-Holland Publishing Company. ISBN: 978-0-7204-2015-9 (cit. on p. 5).

Tip, Frank, Robert M. Fuhrer, Adam Kiezun, Michael D. Ernst, Ittai Balaban, and Bjorn De Sutter (2011). "Refactoring using type constraints." In: *ACM Transactions on Programming Languages and Systems* 33.3, p. 9. DOI: 10.1145/1961204.1961205. (Cit. on pp. 160, 176).

Triplequote (n.d.). *Hydra: The Parallel Scala Compiler*. URL: https://triplequote.com/hydra/compilation/ (cit. on p. 154).

Van Antwerpen, Hendrik, Casper Bach Poulsen, Arjen Rouvoet, and Eelco Visser (2018). "Scopes as types." In: *Proceedings of the ACM on Programming Languages* 2.OOPSLA. DOI: 10.1145/3276484 (cit. on pp. 11, 47, 90–92, 95, 99, 120, 126, 153, 156, 161, 163, 165, 201).

Van Antwerpen, Hendrik and Eelco Visser (2021). "Scope States: Guarding Safety of Name Resolution in Parallel Type Checkers." In: *35th European Conference on Object-Oriented Programming (ECOOP 2021)*. Ed. by Anders Møller and Manu Sridharan. Vol. 194. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 1:1–1:29. DOI: 10.4230/LIPIcs.ECOOP.2021.1. (Cit. on pp. 12, 125, 202).

Van Wijngaarden, Adriaan, B. J. Mailloux, John Edward Lancelot Peck, Cornelis H. A. Koster, Michel Sintzoff, C. H. Lindsey, Lambert G. L. T. Meertens, and R. G. Fisker (1976). *Revised Report on the Algorithmic Language Algol 68*. Springer. DOI: 10.1007/978-3-642-95279-1. (Cit. on p. 4).

van Antwerpen, Hendrik, Pierre Néron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth (2015). *A Constraint Language for Static Semantic Analysis based on Scope Graphs with Proofs*. Tech. rep. TUD-SERG-2015-012. Software Engineering Research Group, Delft University of Technology. URL: http://swerl.tudelft.nl/twiki/pub/Main/TechnicalReports/TUD-SERG-2015-012.pdf (cit. on p. 38).

— (2016). "A constraint language for static semantic analysis based on scope graphs." In: *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*. Ed. by Martin Erwig and Tiark Rompf. ACM, pp. 49–60. DOI: 10.1145/2847538.2847543. (Cit. on pp. 10, 17, 48, 50, 51, 53, 56, 68, 95, 100, 116, 120, 144, 156, 161, 163, 201).

Van Wyk, Eric, Derek Bodin, Jimin Gao, and Lijesh Krishnan (2010). "Silver: An extensible attribute grammar system." In: *Science of Computer Programming* 75.1-2, pp. 39–54. DOI: 10.1016/j.scico.2009.07.004. (Cit. on p. 121).

Visser, Eelco (2015). "Understanding software through linguistic abstraction." In: *Science of Computer Programming* 97, pp. 11–16. DOI: 10.1016/j.scico.2013.12.001. (Cit. on p. 186).

Visser, Eelco, Guido Wachsmuth, Andrew P. Tolmach, Pierre Néron, Vlad A. Vergu, Augusto Passalaqua, and Gabriël Konat (2014). "A Language Designer's Workbench: A One-Stop-Shop for Implementation and Verification of Language Designs." In: *Onward! 2014, Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, part of SPLASH '14, Portland, OR, USA, October 20-24, 2014*. Ed. by Andrew P. Black, Shriram Krishnamurthi, Bernd Bruegge, and Joseph N. Ruskiewicz. ACM, pp. 95–111. DOI: 10.1145/2661136.2661149. (Cit. on pp. 17, 43, 44, 47, 174).

Vytiniotis, Dimitrios, Simon L. Peyton Jones, Tom Schrijvers, and Martin Sulzmann (2011). "OutsideIn(X) Modular type inference with local assumptions." In: *J. Funct. Program.* 21.4-5, pp. 333–412. DOI: 10.1017/S0956796811000098 (cit. on p. 82).

Wachsmuth, Guido, Gabriël Konat, Vlad A. Vergu, Danny M. Groenewegen, and Eelco Visser (2013). "A Language Independent Task Engine for Incremental Name and Type Analysis." In: *Software Language Engineering - 6th International Conference, SLE 2013, Indianapolis, IN, USA, October 26-28, 2013. Proceedings*. Ed. by Martin Erwig, Richard F. Paige, and Eric Van Wyk. Vol. 8225. Lecture Notes in Computer Science. Springer, pp. 260–280. DOI: 10.1007/978-3-319-02654-1_15. (Cit. on pp. 43, 44).

Wenzel, Makarius (2009). "Parallel Proof Checking in Isabelle/Isar." en. In: *ACM SIGSAM Workshop on Programming Languages for Mechanized Mathematics Systems (PLMMS '09)*. New York, NY, USA: Association for Computing Machinery, p. 9 (cit. on p. 155).

— (2013). "Shared-Memory Multiprocessing for Interactive Theorem Proving." In: *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*. Ed. by Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie. Vol. 7998. Lecture Notes in Computer Science. Springer, pp. 418–434. DOI: 10.1007/978-3-642-39634-2_30. (Cit. on pp. 155, 156).

Zhang, Danfeng and Andrew C. Myers (2014). "Toward general diagnosis of static errors." In: *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*. Ed. by Suresh Jagannathan and Peter Sewell. ACM, pp. 569–582. DOI: 10.1145/2535838.2535870. (Cit. on p. 46).

Zhang, Danfeng, Andrew C. Myers, Dimitrios Vytiniotis, and Simon L. Peyton Jones (2015). "Diagnosing type errors with class." In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*. Ed. by David Grove and Steve Blackburn. ACM, pp. 12–21. DOI: 10.1145/2737924.2738009. (Cit. on p. 46).

— (2017). "SHErrLoc: A Static Holistic Error Locator." In: *ACM Transactions on Programming Languages and Systems* 39.4. DOI: 10.1145/3121137. (Cit. on p. 185).

Zhang, Yao and Hao Ma (2014). "Analysis of Networked Control Schemes and Data-Processing Method for Parallel Inverters." In: *IEEE Transactions on Industrial Electronics* 61.4, pp. 1834–1844. DOI: 10.1109/TIE.2013.2267701. (Cit. on p. 185).

Zwaan, Aron (2022). "Specializing Scope Graph Resolution Queries." In: *Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering*. SLE 2022. Auckland, New Zealand: Association for Computing Machinery, pp. 121–133. DOI: 10.1145/3567512.3567523. (Cit. on pp. 182, 184).

Zwaan, Aron and Hendrik van Antwerpen (2023). "Scope Graphs: The Story so Far." In: *Eelco Visser Commemorative Symposium (EVCS 2023)*. Ed. by Ralf Lämmel, Peter D. Mosses, and Friedrich Steimann. Vol. 109. Open Access Series in Informatics (OASIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 32:1–32:13. DOI: 10.4230/OASIcs.EVCS.2023.32 (cit. on pp. 7, 202).

Zwaan, Aron, Hendrik van Antwerpen, and Eelco Visser (2022). "Incremental type-checking for free: using scope graphs to derive incremental type-checkers." In: *Proceedings of the ACM on Programming Languages* 6.OOPSLA2, pp. 424–448. DOI: 10.1145/3563303 (cit. on pp. 182, 184, 202).

Curriculum Vitae

Hendrik van Antwerpen

Education

2002	Gymnasium Guido de Brès, Rotterdam
2011	BEng Computer Science The Hague University of Applied Sciences
2016	MSc Computer Science Delft University of Technology
2025	PhD Computer Science Delft University of Technology

Publications

2016 Hendrik van Antwerpen, Pierre Néron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth (2016). “A constraint language for static semantic analysis based on scope graphs.” In: *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*. DOI: 10.1145/2847538.2847543.

2018 Hendrik van Antwerpen, Casper Bach Poulsen, Arjen Rouvoet, and Eelco Visser (2018). “Scopes as types.” In: *Proceedings of the ACM on Programming Languages* OOPSLA. DOI: 10.1145/3276484

2019 Daniel A. A. Pelsmaeker, Hendrik van Antwerpen, and Eelco Visser (2019). “Towards Language-Parametric Semantic Editor Services Based on Declarative Type System Specifications (Brave New Idea Paper).” In: *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*. Dagstuhl, Germany. DOI: 10.4230/LIPIcs.ECOOP.2019.26

Adrian D. Mensing, Hendrik van Antwerpen, Casper Bach Poulsen, and Eelco Visser (2019). "From definitional interpreter to symbolic executor." In: *Proceedings of the 4th ACM SIGPLAN International Workshop on Meta-Programming Techniques and Reflection*. Athens, Greece. DOI: 10.1145/3358502.3361269

2020 Arjen Rouvoet, Hendrik van Antwerpen, Casper Bach Poulsen, Robbert Krebbers, and Eelco Visser (2020a). "Knowing when to ask: sound scheduling of name resolution in type checkers derived from declarative specifications." In: *Proceedings of the ACM on Programming Languages* OOPSLA. DOI: 10.1145/3428248

2021 Hendrik van Antwerpen and Eelco Visser (2021). "Scope States: Guarding Safety of Name Resolution in Parallel Type Checkers." In: *35th European Conference on Object-Oriented Programming (ECOOP 2021)*. Dagstuhl, Germany. DOI: 10.4230/LIPIcs.ECOOP.2021.1.

2022 Aron Zwaan, Hendrik van Antwerpen, and Eelco Visser (2022). "Incremental type-checking for free: using scope graphs to derive incremental type-checkers." In: *Proceedings of the ACM on Programming Languages* OOPSLA2. DOI: 10.1145/3563303

Daniël A. A. Pelsmaeker, Hendrik van Antwerpen, Casper Bach Poulsen, and Eelco Visser (2022). "Language-parametric static semantic code completion." In: *Proceedings of the ACM on Programming Languages* OOPSLA. DOI: 10.1145/3527329.

2023 Aron Zwaan and Hendrik van Antwerpen (2023). "Scope Graphs: The Story so Far." In: *Eelco Visser Commemorative Symposium (EVCS 2023)*. Dagstuhl, Germany. DOI: 10.4230/OASIcs.EVCS.2023.32

Douglas A. Creager and Hendrik van Antwerpen (2023). "Stack Graphs: Name Resolution at Scale." In: *Eelco Visser Commemorative Symposium (EVCS 2023)*. Dagstuhl, Germany. DOI: 10.4230/OASIcs.EVCS.2023.8.

0000-0001-5117-0921

Titles in the IPA Dissertation Series since 2022

A. Fedotov. *Verification Techniques for xMAS*. Faculty of Mathematics and Computer Science, TU/e. 2022-01

M.O. Mahmoud. *GPU Enabled Automated Reasoning*. Faculty of Mathematics and Computer Science, TU/e. 2022-02

M. Safari. *Correct Optimized GPU Programs*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2022-03

M. Verano Merino. *Engineering Language-Parametric End-User Programming Environments for DSLs*. Faculty of Mathematics and Computer Science, TU/e. 2022-04

G.F.C. Dupont. *Network Security Monitoring in Environments where Digital and Physical Safety are Critical*. Faculty of Mathematics and Computer Science, TU/e. 2022-05

T.M. Soethout. *Banking on Domain Knowledge for Faster Transactions*. Faculty of Mathematics and Computer Science, TU/e. 2022-06

P. Vukmirović. *Implementation of Higher-Order Superposition*. Faculty of Sciences, Department of Computer Science, VU. 2022-07

J. Wagemaker. *Extensions of (Concurrent) Kleene Algebra*. Faculty of Science, Mathematics and Computer Science, RU. 2022-08

R. Janssen. *Refinement and Partiality for Model-Based Testing*. Faculty of Science, Mathematics and Computer Science, RU. 2022-09

M. Laveaux. *Accelerated Verification of Concurrent Systems*. Faculty of Mathematics and Computer Science, TU/e. 2022-10

S. Kochanthara. *A Changing Landscape: On Safety & Open Source in Automated and Connected Driving*. Faculty of Mathematics and Computer Science, TU/e. 2023-01

L.M. Ochoa Venegas. *Break the Code? Breaking Changes and Their Impact on Software Evolution*. Faculty of Mathematics and Computer Science, TU/e. 2023-02

N. Yang. *Logs and models in engineering complex embedded production software systems*. Faculty of Mathematics and Computer Science, TU/e. 2023-03

J. Cao. *An Independent Timing Analysis for Credit-Based Shaping in Ethernet TSN*. Faculty of Mathematics and Computer Science, TU/e. 2023-04

K. Dokter. *Scheduled Protocol Programming*. Faculty of Mathematics and Natural Sciences, UL. 2023-05

J. Smits. *Strategic Language Workbench Improvements*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2023-06

A. Arslanagić. *Minimal Structures for Program Analysis and Verification*. Faculty of Science and Engineering, RUG. 2023-07

M.S. Bouwman. *Supporting Railway Standardisation with Formal Verification*. Faculty of Mathematics and Computer Science, TU/e. 2023-08

S.A.M. Lathouwers. *Exploring Annotations for Deductive Verification*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2023-09

J.H. Stoel. *Solving the Bank, Lightweight Specification and Verification Techniques for Enterprise Software*. Faculty of Mathematics and Computer Science, TU/e. 2023-10

D.M. Groenewegen. *WebDSL: Linguistic Abstractions for Web Programming*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2023-11

D.R. do Vale. *On Semantical Methods for Higher-Order Complexity Analysis*. Faculty of Science, Mathematics and Computer Science, RU. 2024-01

M.J.G. Olsthoorn. *More Effective Test Case Generation with Multiple Tribes of AI*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2024-02

B. van den Heuvel. *Correctly Communicating Software: Distributed, Asynchronous, and Beyond*. Faculty of Science and Engineering, RUG. 2024-03

H.A. Hiep. *New Foundations for Separation Logic*. Faculty of Mathematics and Natural Sciences, UL. 2024-04

C.E. Brandt. *Test Amplification For and With Developers*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2024-05

J.I. Hejderup. *Fine-Grained Analysis of Software Supply Chains*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2024-06

J. Jacobs. *Guarantees by construction*. Faculty of Science, Mathematics and Computer Science, RU. 2024-07

O. Bunte. *Cracking OIL: A Formal Perspective on an Industrial DSL for Modelling Control Software*. Faculty of Mathematics and Computer Science, TU/e. 2024-08

R.J.A. Erkens. *Automaton-based Techniques for Optimized Term Rewriting*. Faculty of Mathematics and Computer Science, TU/e. 2024-09

J.J.M. Martens. *The Complexity of Bisimilarity by Partition Refinement*. Faculty of Mathematics and Computer Science, TU/e. 2024-10

L.J. Edixhoven. *Expressive Specification and Verification of Choreographies*. Faculty of Science, OU. 2024-11

J.W.N. Paulus. *On the Expressivity of Typed Concurrent Calculi*. Faculty of Science and Engineering, RUG. 2024-12

J. Denkers. *Domain-Specific Languages for Digital Printing Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2024-13

L.H. Applis. *Tool-Driven Quality Assurance for Functional Programming and Machine Learning*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2024-14

P. Karkhanis. *Driving the Future: Facilitating C-ITS Service Deployment for Connected and Smart Roadways*. Faculty of Mathematics and Computer Science, TU/e. 2024-15

N.W. Cassee. *Sentiment in Software Engineering*. Faculty of Mathematics and Computer Science, TU/e. 2024-16

H. van Antwerpen. *Declarative Name Binding for Type System Specifications*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2025-01