Language-Parametric Methods for Developing Interactive Programming Systems

Konat, Gabriël

DOI
10.4233/uuid:03d70c5d-596d-4c8c-92da-0398dd8221cb

Publication date
2019

Document Version
Final published version

Citation (APA)
Konat, G. (2019). *Language-Parametric Methods for Developing Interactive Programming Systems*. [Dissertation (TU Delft), Delft University of Technology]. https://doi.org/10.4233/uuid:03d70c5d-596d-4c8c-92da-0398dd8221cb

Gabriël Ditmar Primo Konat was born in The Hague, the Netherlands. In 2009, he received his BSc in Computer Science from the Institute of Applied Sciences in Rijswijk. In 2012, he received his MSc in Computer Science from Delft University of Technology (TUDelft). From 2012 to 2018, he was a Ph.D. student with the Programming Languages group at TUDelft, under supervision of Eelco Visser and Sebastian Erdweg. His work focuses on language workbenches and incremental build systems.

TUDelft

Language-Parametric Methods for Developing Interactive Programming Systems Gabriël Konat

Language-Parametric Methods for Developing Interactive Programming Systems

Gabriël Konat

Propositions

accompanying the dissertation

Language-Parametric Methods for Developing Interactive Programming Systems

by

Gabriël Ditmar Primo Konat

1. Language-parametric methods for developing interactive programming systems are feasible and useful. (This dissertation)
2. Compilers of general-purpose languages must be bootstrapped with fixpoint bootstrapping. (This dissertation)
3. Manually implementing an incremental system must be avoided. (This dissertation)
4. Like chemists need lab assistants, computer scientists need software engineers to support them in research, teaching, and application in industry.
5. Programming languages that evolve via public request for comments (RFCs) attract a diverse range of people, and are therefore of higher quality.
6. Critical case studies are a valuable tool for providing evidence in research.
7. Developing an interactive video game is the most effective way to learn a new programming language.
8. The publication process of conferences with a yearly deadline and unidirectional feedback is not conducive to innovative and high-quality publications.
9. Rewriting a C or C++ program in Rust always increases code quality.

These propositions are regarded as opposable and defendable, and have been approved as such by the promotors prof.dr. E. Visser and prof.dr. S.T. Erdweg.

Language-Parametric Methods for Developing Interactive Programming Systems

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 for Doctorates
to be defended publicly on
Monday 18 November 2019 at 15:00 o'clock
by

Gabriël Ditmar Primo KONAT

MSc Computer Science, Delft University of Technology, the Netherlands
born in The Hague, the Netherlands

This dissertation has been approved by the promotors.

Composition of the doctoral committee:

Rector Magnificus,	chairperson
Prof.dr. E. Visser	Delft University of Technology, promotor
Prof.dr. S.T. Erdweg	Johannes Gutenberg University Mainz, promotor

Independent members:

Prof.dr.ir. D.H.J. Epema	Delft University of Technology
Prof.dr. M. Flatt	University of Utah
Prof.dr. T. van der Storm	University of Groningen / CWI
Dr. A. Mokhov	Newcastle University
Dr. E. Dolstra	Tweag I/O
Prof.dr. K.G. Langendoen	Delft University of Technology, reserve member

The work in this dissertation has been carried out at the Delft University of Technology, and was supported by NWO/EW Free Competition Project 612.001.114 (Deep Integration of Domain-Specific Languages).

Cover: Museum of Pop Culture - Photo © 2014 Gabriël Ditmar Primo Konat

Printed and bound by: Gildeprint - https://www.gildeprint.nl/

ISBN: 978-94-6366-210-9

Contents

Samenvatting

Op alle computers wordt *software* uitgevoerd, zoals besturingssystemen, webbrowsers, en videospellen, die door miljarden mensen over de wereld worden gebruikt. Daarom is het belangrijk om software van hoge kwaliteit te bouwen, wat alleen mogelijk is met *interactive programmeringssysteemen* die programmeurs betrekken in de uitwisseling van *correcte* en *responsieve* feedback. Gelukkig maken geïntegreerde software-ontwikkelingsomgevingen dit mogelijk voor vele generieke programmeertalen, door middel van broncodebewerkers met hulpmiddelen zoals syntaxiskleuring en automatische aanvulling.

Daarintegen zijn *domeinspecifieke talen* programmeertalen die gespecialiseerd zijn voor een specifiek probleemdomein, en het daarom mogelijk maken om betere software te schrijven door directe expressie van problemen en oplossingen in termen van het domein. Echter, omdat domeinspecifieke talen gespecialiseerd zijn voor een bepaald domein, en er veel probleemdomeinen zijn, moeten we veel nieuwe domeinspecifieke talen ontwikkelen, inclusief bijbehorende interactieve programmeringssysteemen!

Het ontwikkelen van een ad-hoc interactief programmeringssysteem voor een domeinspecifieke taal is ondoenlijk, omdat dit een te grote ontwikkelingsinspanning nodig heeft. Daarom is het onze visie om *taalparametrische methodes* voor het ontwikkelen van interactieve programmeringssysteemen te gebruiken. Een taalparametrische methode neemt als invoer een beschrijving van een domeinspecifieke taal, en implementeert automatisch (delen van) een interactief programmeringssysteem, waardoor ontwikkelingsinspanning wordt verminderd, en domeinspecifieke taalontwikkeling doenlijk wordt. In dit proefschrift ontwikkelen we drie taalparametrische in de vijf kernhoofdstukken.

We ontwikkelen een taalparametrische methode voor incrementele naam- en type-analyse, waarbij taalontwikkelaars de naam- en typeregels van hun domeinspecifieke taal specificeren in metatalen (talen die gespecialiseerd zijn in het domein van taalontwikkeling). Uit een dergelijke specificatie leiden we automatisch een incrementele naam- en typeanalyse af, inclusief bewerkingshulpmiddelen zoals codeaanvulling en inline-foutmeldingen.

We ontwikkelen een taalparametrische methode voor het interactief bootstrappen van de metataal compilers van taalwerkbanken. We beheren meerdere versies van metataal compilers, geven expliciet de afhankelijkheden tussen hen aan, en voeren fixpoint bootstrapping uit, waarbij we iteratief metataal compilers op henzelf toepassen om nieuwe versies af te leiden totdat er geen verandering plaatsvindt, of totdat een fout wordt gevonden. Deze bootstrappingbewerkingen kunnen worden gestart en teruggedraaid (wanneer een fout is gevonden) in het interactieve programmeringssysteem van de taalwerkbank.

Ten slotte ontwikkelen we Pipelines for Interactive Environments (PIE), een parametrische methode voor het ontwikkelen van interactieve pijplijnen voor softwareontwikkeling, een superset van interactieve programmeeromgevingen.

Met PIE kunnen pijplijnontwikkelaars bondig pijplijnprogramma's schrijven in termen van taken en afhankelijkheden tussen taken en bestanden, die PIE vervolgens incrementeel uitvoert. PIE schaalt af naar vele veranderingen met kleine impact, en schaalt op naar grote afhankelijkheidsgrafieken via een incrementeel veranderingsgedreven algoritme.

Summary

All computers run *software*, such as operating systems, web browsers, and video games, which are used by billions of people around the world. Therefore, it is important to develop high-quality software, which is only possible through *interactive programming systems* that involve programmers in the exchange of *correct* and *responsive* feedback. Fortunately, for many general-purpose programming languages, integrated development environments provide interactive programming systems through code editors and editor services.

On the other hand, Domain-Specific Languages (DSLs) are programming languages that are specialized towards a specific problem domain, enabling better software through direct expression of problems and solutions in terms of the domain. However, because DSLs are specialized to a specific domain, and there are many problem domains, we need to develop many new DSLs, including their interactive programming systems!

Ad-hoc development of an interactive programming system for a DSL is infeasible, as developing one requires a huge development effort. Therefore, our vision is to create and improve *language-parametric methods* for developing interactive programming systems. A language-parametric method takes as input a description of a DSL, and automatically implements (parts of) an interactive programming system, reducing development effort, thereby making DSL development feasible. In this dissertation, we develop three language-parametric methods throughout the five core chapters.

We develop a language-parametric method for incremental name and type analysis, in which language developers *specify* the name and type rules of their DSL in meta-languages (languages specialized towards the domain of language development). From such a specification, we automatically derive an incremental name and type analysis, including editor services such as code completion and inline error messages.

We develop a language-parametric method for interactively bootstrapping the meta-language compilers of language workbenches. We version meta-language compilers, explicitly denote dependencies between them, and perform *fixpoint bootstrapping*, where we iteratively self-apply meta-language compilers to derive new versions until no change occurs, or until a defect is found. These bootstrapping operations can be started and rolled back (when defect) in the interactive programming system of the language workbench.

Finally, we develop PIE, a parametric method for developing interactive software development pipelines, a superset of interactive programming environments. With PIE, pipeline developers can concisely write pipeline programs in terms of tasks and dependencies between tasks and files, which the PIE runtime then incrementally executes. PIE scales down to many low-impact changes and up to large dependency graphs through a change-driven incremental build algorithm.

Preface

My journey to this dissertation started when I was two years old. My grandpa, who knew a lot about computers because he was a computer technician, would regularly let me play on his computer when we visited. At first, he'd teach me to play simple point and click games, painting programs, and colouring book programs. Later on he'd teach me DOS commands, how to navigate in Windows, and – more importantly – how to play SimCity 2000. I of course did not fully understand SimCity at the time, but enjoyed it a lot nonetheless, because it would let me create and destroy cities, and hear 'bzzt' a million times. I would usually run out of money pretty fast and have to take several bonds, which quickly lead to bankruptcy. Eventually, I did manage to complete a scenario which rewarded me the key of the city!

My grandpa was also one of the first to have cable internet in the Netherlands, which had a fixed monthly cost in contrast to the by-minute cost of dialup internet, making it a lot more affordable, even though the 64Kb/s speed was horrible. The internet was really a magical experience back then, because it enabled easily chatting with people, finding news, information, games, cheats, jokes, or basically anything. Every day, you'd find something new and exiting.

When visiting my grandparents, I would quickly jump behind the computer and go on the internet, usually playing free online games. One online game that stood out is Graal Online, an MMORPG that plays like Zelda: A Link to the Past, which is still alive as of writing this dissertation. While the online mode of Graal Online is a lot of fun, it also features an offline mode with a level editor, allowing you to create your own worlds. The level editor also includes a script editor with a Java-like scripting language, which was my first foray into programming. I was able to take the offline mode home by compressing it and splitting it across 12 floppy disks, allowing me to build worlds and program from my own computer, which I frequently did. Together with a friend, we designed and programmed our own world and submitted it to the creator of Graal Online, asking for it to be hosted as a 'playerworld'. Unfortunately, we never got a response.

In 2005, I completed my secondary education at senior (Dutch: HAVO) level. Because of my grandpa's enthusiasm for computers, teaching me how to use them, letting me regularly use his computer and internet, and learning to program with Graal Online, programming computers became my vocation. However, to be able to go to a university, I'd have to follow two more years of university preparatory education (VWO). Instead I opted to do the more practically-oriented higher professional education (hoger beroepsonderwijs/HBO) at the Institute of Applied (Computer) Sciences in Rijswijk, as I did not want to spend two more years in secondary education, and instead wanted to immediately specialise in software development. I'm very glad to have chosen this option, as I got to meet many like-minded students, learn about

how computers work, learn to properly program in several real programming languages, and learn how to develop software.

In my last year, I was still looking for a project to graduate on. As usual, I was procrastinating to the very end and almost got into trouble for not having a graduation project. Luckily, during one of the robotics labs, Martijn Wisse from Delft University of Technology (TUDelft) visited and advertised one of his graduation projects: building a vision system for their autonomous humanoid football robot, which would compete in RoboCup 2009, Graz, Austria. Together with fellow student Jonathan Staats, we of course agreed to do this awesome graduation project. We'd work with the team in the basement of the TUDelft 3mE building (of course they put a robotics team in the basement) to build the eyes of their robot TUlip, supervised by Boris Lenseigne.

The vision system consisted of two small Linux-based boards with cameras, connected to the robot's main computer to provide stereo information. Jonathan worked on writing a Linux device driver for the camera, while I worked on the software that uses the camera data to detect field lines (Hough transform) and the ball. TUlip's main computer would then integrate this data to determine its position and the position of the ball.

TUlip competed in the Robocup and was able to defend the goal, walk short distances, kick the ball, and track the ball through our vision system. However, one of the challenges was to find the ball, dribble it, and then kick it to score, which turned out to be really hard for our big and heavy bipedal robot with a realistic humanoid gait. In any case, RoboCup was an awesome experience, with lots of great football matches with smaller and wheeled robots, lots of like-minded people, and a lot of fun with our team. In the end, despite not winning RoboCup, we were able to graduate and receive the title of engineer (ing. in Dutch). Besides graduating, this project gave me a very high appreciation of TUDelft and the people working there. The university is far more practical than I had imagined, and most students/staff are very kind, open-minded, and motivated, convincing me to study for a master's degree at TUDelft.

While studying for a master's degree, I would regularly listen to podcasts on my bus trip from The Hague to Delft. One of the podcasts I was listing to at the time was the Software Engineering Radio (SE Radio), founded by Markus Voelter. Episode 118 was on parsing, with an interview of Eelco Visser, which surprised me because I recognised him as the lecturer of Delft's Compiler Construction course. I found the episode quite interesting, as I really like programming, and building a parser for a (new?) programming language had always interested me, but I never quite understood it or had the time/motivation to dive into it. I thought it was also quite cool that a professor at my university was being interviewed in my favourite podcast.

In the first semester of the second year, I did the compiler construction course. I would have done it in the first year, but everyone I talked to at the introductory week (even other teachers!) recommended me to not do it in the first year, because it was a really hard course that required a lot of time investment (a lot more than you'd get ECTS for). While this was definitely true, I would have liked and be motivated by the course so much that it wouldn't

have been a problem at all.

In any case, I followed the course, but was slightly discontented. Even though Eelco was the responsible teacher, one of his postdocs, Guido Wachsmuth, was actually teaching the course, so I'd never meet Eelco during the course. Note that this is not a jab at Guido at all, because he did a great job teaching compiler construction. He is very knowledgable, has great-looking slides with a lot of information, and is very supportive of his students. That is, the students that came to the lecture on time at least, as he would usually lock the door when the lecture started to prevent annoyance by late students. I got locked out of half a lecture once because of a flat tire on my bike, oh well. I particularly like Guido's teaching style of explaining algorithms by showing a lot of examples and actually executing the algorithm step-by-step on the slides, which my brain really appreciates. If you'd want to see the algorithm pseudocode, you can just read the corresponding book or paper.

Besides the lectures, there is also a lab in which you build a compiler and IDE from scratch for the MiniJava programming language. To achieve this, we use the Spoofax Language Workbench, which is basically a set of tools to develop programming languages and their interactive programming systems, which is where part of this dissertation's title comes from. Spoofax was first developed by Karl Trygve Kalleberg as an Eclipse IDE for the Stratego language, but was later developed into a Language Workbench by Lennart Kats, who would sometimes visit the lab (in his cool leather jacket) to help us out. While Spoofax was a bit janky sometimes, as many research tools are, it did enable us to develop a compiler and editor for a programming language – starting without any compiler construction knowledge – in a single semester, how cool is that? This lab got me really enthusiastic about developing programming languages, but also about creating meta-tools (i.e., language-parametric methods) for developing interactive programming systems.

In the follow-up course, Model Driven Software Development, which was actually taught by Eelco Visser, we used our knowledge from the compiler construction course to develop our own domain-specific language (DSL). Again together with Jonathan, we constructed a DSL for our 3D virtual world/game engine, Diversia, which we developed during our time at Rijswijk. It was an event-driven DSL that reacted to events in the game world, and compiled down to LUA scripting code which interacted with the game engine. During the course, there was a guest lecture by Markus Voelter, whose podcast sparked my interest for programming languages in the first place, reminding me of how small the world usually is. It was also quite cool to see Markus after hearing him interview so many people.

The final programming languages related course was a seminar on meta-programming, where we dived into program analysis with Datalog. After this course, I was interested in doing my master's thesis with the programming languages group, and Eelco agreed to supervise me. Lennart and Karl got me up to speed with Spoofax development in their office. I sat next to Karl for a couple of days, which, now that I think of it, was probably quite annoying for him since I took up half of his desk and invaded his privacy. I moved onto a

separate table in their office after a while, which was still quite nice compared to the separate master student room (Dutch: het master hok). Later on, Guido became more heavily involved with my thesis, as the topic shifted to name analysis, and I ended up doing my master's thesis on "Language-Parametric Incremental and Parallel Name Resolution".

Before graduating, Eelco asked if I wanted to do a PhD. I knew that doing a PhD was an option, but never really gave it much thought, as I was planning to take a couple of months off and then go into industry. However, after a couple of days, I accepted his offer, and the work on this dissertation begun.

Acknowledgements. Before continuing, I would like to (try to) thank everyone that made this work possible. First, I want to thank my supervisors.

I am grateful to Eelco Visser for giving me the opportunity to do a PhD, and his supervision and kindness throughout it. Eelco is extremely good at understanding the bigger picture. When approached with a problem, he will effortlessly break it down into smaller more understandable parts, a path to solving it, and how to present it to a larger audience. This was invaluable during my PhD, as I learned how to structure my research, writing, and presentations. Eelco is also very kind, understanding, and motivating, even when things went wrong in research or real life, which would happen from time to time. Besides that, we would regularly discuss ideas, research, and applications that we would like to do. Since Eelco hired me as a Postdoctoral researcher, we will try to complete more of that.

I would like to thank Guido Wachsmuth for his supervision, guidance, and kindness in the first half of my PhD, in which we worked closely together to develop declarative and incremental name and type analysis. Guido has a knack for deeply understanding a topic and explaining it in a very approachable manner, which was extremely useful at the start of my PhD. I learned many things from Guido, including compiler construction, (meta-)domain-driven design, Stratego programming, proper benchmarking, and academic writing. Besides that, we always had a lot of interesting ideas and conversations, and in general just had a lot of fun. Guido has found a new opportunity at Oracle Labs as a member of the PGX team, but we still regularly talk and cooperate, as some of our research is being applied at Oracle Labs.

I wish to thank Sebastian Erdweg for his supervision in the latter half of my PhD, in which we worked on bootstrapping and incremental build systems. Without Sebastian's guidance, the research would have never finished on time, or be of the quality that it is now. Sebastian always gave extremely good feedback on my planning, writing, and research, which helped me to improve those skills a lot, especially academic writing. One of Sebastian's superpowers is coming up with interesting examples (that may make or break your approach) within seconds, which was extremely useful for developing incremental build systems as it has many nasty corner cases.

I am grateful to Michael Steindorfer for his help on our interactive pipeline research, moments after finishing his PhD on data structures. He was able to quickly jump into our research and help out, and came up with the name PIE, which stands for Pipelines for Interactive Environments, which we still use

today.

Hassan Chafi gave me the opportunity to do three summer traineeships at Oracle Labs in California, for which I am grateful. I was able to learn a great deal about practical application of my research in a more corporate setting, and got to meet many like-minded people. Besides the great work environment, having the opportunity to live in California and explore it during my free time was wonderful. The weather in California, in contrast to the Netherlands, was always sunny (except for the San Francisco fog) and dry, which made exploring cities and nature delightful.

I would like to thank the members of the committee, Dick Epema, Matthew Flatt, Tijs van der Storm, Andrey Mokhov, Eelco Dolstra, and Koen Langendoen for reviewing my dissertation.

I have many past and current colleagues to thank for their help and hospitality. Danny Groenewegen co-manages the servers that run our build farm and artifact server that greatly increase the productivity of our group. He also co-authored the task engine paper by helping us redefine the name binding and type system of WebDSL, which we used as a subject in our evaluation. Besides that, Danny is very open-hearted, always welcoming newcomers by inviting them to activities. I got into Magic the Gathering and Factorio because Danny invited me to his friend group, which I still play with to this day.

Vlad Vergu worked on several important parts of our infrastructure. At the start of his PhD, he converted and moved the existing SVN repository to GitHub, which greatly increased our productivity. Vlad bootstrapped the Java version of the Stratego compiler, which was previously bootstrapped with a fixed baseline C version of the Stratego compiler, enabling further development on the Java version. He also built a command-line version of Spoofax, called Sunshine, which was an inspiration for Spoofax Core, a platform-independent version of Spoofax. Finally, he co-authored the task engine paper by performing the majority of the benchmarking, which was invaluable to the evaluation in that paper.

Lennart Kats and Karl Trygve Kallenberg got me started with Spoofax and my PhD in general. They built Spoofax and several other related tools that made this line of research possible. Lennart's dissertation on Building Blocks for Language Workbenches inspired parts of this dissertation. Karl came up with the name Spoofax, which consists of Spoo (a food from Babylon 5) and fax (as in a fax machine), for which the domain-name was free, and which is easy to search for.

Maartje de Jonge had just finished her PhD on Language-Parametric Techniques for Language-Specific editors, which also inspired parts of this dissertation. Because Maartje just finished when I started, she was leaving Delft and was able to transfer her apartment to me, finally giving me an affordable place to stay in Delft.

Hendrik van Antwerpen started out as a programmer in our group, helping immensely in getting Spoofax Core to work. He developed a Maven plugin, implemented cross-language dependencies, bootstrapped the meta-languages, and made several improvements make Spoofax more cross-platform. Later he

did his master thesis with us, and then started his PhD, taking over the name and type analysis work. I admire Hendrik's ability to deeply understand and continuously improve large systems.

Luis Eduardo de Souza Amorim (Eduardo) started his PhD roughly the same time as I did. He has worked on improved the parsing and editor services side of Spoofax. Eduardo's hard work, perseverance, and ability to understand things is something I greatly admire. He was able to dive into the existing parser generator and parser code, understand every interaction, and make many very important improvements to it, all while publishing academic papers.

Daco Harkes evaluated Spoofax during his PhD by developing his IceDust DSL in it, which drove us to continuously improve Spoofax. We also always had interesting conversations about our research and a whole range of other (probably more geeky) topics. I admire Daco's ability to go very broad, take in all available information, come to a decision and explain it, and then forge his own path forward.

Daniel Pelsmaeker evaluated Spoofax Core in his master thesis by building an IntelliJ plugin for it, showing that it is indeed cross-platform. He made several improvements to Spoofax Core to make it more platform-independent, including a cross-platform configuration framework. He now started a PhD with us on editor services. Daniel is always enthusiastic and optimistic, and has a very fine attention to detail, which I appreciate a lot.

Jeff Smits developed FlowSpec, a new meta-DSL for Spoofax for declaratively specifying dataflow analyses, and developed an incremental Stratego compiler, which can speed up the compilation time of a Spoofax project almost by an order of magnitude. He has also made several contributions to Spoofax and PIE. I admire Jeff's ability to absorb knowledge and to be able to easily explain it to others, and his continuous drive to improve our tools and code.

Jasper Denkers engineered JSGLR2, a replacement for the parser of Spoofax, with a modular architecture and better performance. He has also made several improvements to the build system and modularity of Spoofax. In his PhD, he is applying and evaluating Spoofax to real-world DSLs in industry. Jasper's ability to decompose a system into its constituents and improve it, and his ability to connect our research to industry, is something I admire.

Martijn Dwars set up automated feedback and grading for the Compiler Construction lab, which greatly reduced the workload for graders, while giving students the opportunity to get feedback early and improve their grade. He also contributed numerous small improvements to Spoofax. Martijn is one of the best programmer and problem solver I know.

I wish to thank Elmer van Chastelet and Stepen van der Laan (along with Danny) for managing our servers and keeping them working. I am grateful to Roniet Sharabi, secretary of our group, who is always supportive, has helped me understand the bureaucracy of the university numerous times, and always made sure the academic process went smoothly. Last but not least, I would like to thank Sven Keidel, Peter Mosses, Casper Bach Poulsen, Robbert Krebbels, Arjen Rouvoet, Tamás Szabó, Paolo Giarrusso, Roelof Sol,

Volker Lanting, Oskar van Rest, Taico Aerts, and Chiel Bruin. We had many interesting conversations and it was a pleasure working with you.

Finally, I would like to thank my friends and family. This work would not have been possible without my parents, Helma and Rasit, who have always been there for me and supported me my entire life. I would like to thank my grandpa, Loe, for always being there for me and teaching me how to use computers since I was two years old, which led to programming being my vocation. Unfortunately, he passed away in 2013. While he cannot be here in person, he will always be in my heart. I am grateful to Charlotte for always being there for me to talk, keeping me (somewhat) sane through this PhD adventure. Finally, I'd like to thank the rest of my friends and family, of which there are too many to write down, for their support.

Introduction

1

My thesis is that language-parametric methods for developing interactive programming systems are feasible and useful.

All computers run *software*, such as operating systems, web browsers, chat applications, photo editors, and video games. Software is used on many different computer systems by billions of people around the world, and has become such a crucial part of our lives. Therefore, it is important that we develop high-quality software.

Software consists of *programs* that control computers, which are developed using *programming languages*. Typically, these programming languages are General-Purpose Languages (GPLs), supporting the development of many different kinds of software, applicable across many problem domains. To develop high-quality software, we do not only need good programming languages and programmers, but also need high-quality *interactive programming systems* that involve programmers in the exchange of *correct* and *responsive* feedback. Fortunately, for many GPLs, Integrated Development Environments (IDEs) provide correct and responsive interactive programming systems through code editors, editor services, and inline feedback.

On the other hand, Domain-Specific Languages (DSLs) are programming languages that are specialized towards a specific problem domain. DSLs allow us to develop better software through linguistic abstraction for specific problem domains, supporting direct expression of problems and solutions in terms of the domain, and domain-specific constraint checking. However, because DSLs are specialized towards a specific domain, and there are many problem domains, we need to develop many new DSLs, including their interactive programming systems!

Manually developing an interactive programming system for a DSL is not feasible, as developing one requires a huge development effort [77]. Therefore, our vision is to create and improve *language-parametric methods* for developing interactive programming systems. A language-parametric method takes as input a description of a language, and automatically implements (parts of) an interactive programming system, reducing the development effort, thereby making DSL development feasible.

We now explore this vision in detail in the rest of this introductory chapter. We cover background information on programming languages, domain-specific languages, and programming systems. We describe interactive programming systems and the challenges in developing them. We describe our vision to tackle these challenges: language-parametric methods for developing interactive programming systems. We summarize our contributions, which show the feasibility and usefulness with concrete instances of these language-parametric methods. Finally, we describe our research methodology and the structure of

the rest of the dissertation.

1.1 Programming Systems

A *programming language* is a formal language for writing *programs* that control computers. Programmers, or software developers, develop software that consist of programs or source code written in one or more programming languages. Therefore, programs are a mechanism to communicate the intent of a programmer to a computer.

Programming languages such as C, Java, and Rust, are *General-Purpose Languages (GPLs)*, supporting the development of many different kinds of software, applicable across many problem domains. New general-purpose programming languages appear less often since developing, evolving, and maintaining one is a large undertaking.

On the other hand, *Domain-Specific Languages (DSLs)* are programming languages that are specialized to a specific domain, supporting only the development of solutions to the problem domain [27]. Examples of DSLs are Pic, a language for specifying diagrams in terms of boxes and arrows [11]; Structured Query Language (SQL) [23], a language to declaratively query, modify, and compute data in relational databases; Make [133], a declarative language to specify file-based build systems with incremental execution; and Backus–Naur Form (BNF) [10, 82], a meta-DSL (i.e., a DSL specializing in the domain of languages) to declaratively specify context-free grammars of programming languages.

Domain-specific languages provide several advantages over their general-purpose counterparts. Because a DSL is specialized towards a single domain, it can support direct expression of problems and solutions in terms of the domain. Furthermore, because DSL programs relate directly to the problem domain, a DSL can provide domain-specific syntax and perform domain-specific error checking, statically ruling out wrong programs that could not be restricted by a GPL. Finally, because a DSL program is of a higher level than a GPL program, it is possible to derive multiple semantics from a single DSL program by different interpretations or compilation schemata. However, because DSLs are specialized to a particular domain, and there are many problem domains, we need many DSLs, and they need to be developed [102] and maintained [26].

A *programming system* is needed when developing a new language, consisting at least of a *compiler* that validates programs and transforms them into an executable form [2]. For example, the Java programming system consists of the `javac` compiler which checks and transforms Java source files into Java bytecode Intermediate Representation (IR). The Rust programming system has the `rustc` compiler which compiles rust source files to machine code. Besides a batch compiler, programming systems typically contain more tooling such as package managers, build systems, interpreters, and debuggers. For example, Rust has the `cargo` package manager. Java has the Java Virtual Machine (JVM) which executes Java bytecode IR. However, we will focus on a compiler.

Programmers interact with programming systems through a command-line terminal, manually running the batch compiler after they have made changes

to the source code, providing feedback to the programmer in the form of compiler error messages and warnings for invalid programs, or lack thereof for valid programs. While this is a flexible way to develop programs – a programmer can choose to use any source code editor, and run the batch compiler on any operating system in their favorite shell – this development process suffers from several problems.

First of all, there is a disconnect between the programming system and the source code editor: error messages from the batch compiler are not displayed inline, and instead need to be manually traced from the text in the terminal back to the source code, introducing a cognitive gap. Furthermore, batch compilers are typically not responsive: the programmer needs to manually run the batch compiler after making changes to the source code, and wait for feedback, inducing a slow and tedious feedback cycle. Finally, there is a lack of feedback: batch compilers only provide error/warning messages. Editor services such as syntax highlighting, structure outlines, or code completion are missing, because the command-line terminal is restricted to text.

These problems limit programmer productivity. We need an *interactive programming system* that increases productivity by providing automatic, continuous, and inline feedback to the programmer.

1.2 Interactive Programming Systems

An interactive programming system is a programming system that is designed to involve the user in the exchange of information [68]. In an interactive programming system, there is a continuous exchange of information between the programmer and the system: a *feedback cycle* where the programmer edits the source code of a program, the system sends back feedback, and so forth.

Integrated Development Environments (IDEs) such as Atom, Eclipse, emacs, IntelliJ, MPS, NetBeans, Notepad++, vim, Visual Studio (Code), and XCode are interactive programming systems for certain programming languages. For example, IntelliJ IDEA is an interactive programming system including built-in support for Java and Kotlin. Interactive programming systems increase productivity by:

- Closing the cognitive gap by providing inline error/warning messages and other interactions directly in terms of the source code through a code editor. For example, regions in the source code with errors are highlighted, show the error message when hovered with the mouse, and support application of quick fixes which directly modify the source code.
- Automatically providing feedback when changes to the source code are made, and providing this feedback in a timely manner.
- Providing better feedback in the form of editor services. For example, syntax coloring provides typographical styling based on the syntactical and semantic meaning of code, structure outlines provide browsable summaries of the source code, reference resolution supports browsing between declarations and references, and code completion provides context-dependent browsing

and automatic completion of unambiguous source code sentences [114].

However, for a programming system to be truly interactive, it must be *correct* and *responsive*. An interactive programming system only provides feedback that is correct, when the process providing that feedback is precise. A responsive interactive programming system provides feedback automatically: without explicit user interaction where possible, and more importantly, in a timely manner.

When an interactive programming system lacks these qualities, productivity is lost. For example, providing an incorrect code completion or quick fix to the programmer, which leads to errors in the code after application, confuses and annoys the programmer. Furthermore, explicitly having to ask for syntax coloring, or waiting for five seconds to get a new structure outline, after modifying the source code, is tedious.

A method to achieve responsiveness in interactive programming systems is *incrementality*, where the response time is proportional to the impact of a change to the source code. An incremental system achieves this by only recomputing outputs that have been affected by a changed input, while reusing previously computed outputs. For example, typing a character into a code editor (most of the time) does not affect the syntax coloring of the text before and after that new character, requiring only syntax coloring of the newly typed character.

However, responsiveness is only achieved when incrementality is *scalable*, where the programming system can *scale down* to many low-impact source code changes, while *scaling up* to large programs. Since most source code changes have a low impact, few outputs should be recomputed, and response times should be fast, even though programs are large.

Finally, it is important that correctness is still guaranteed in the presence of incrementality and scalability. Developing correct and responsive interactive programming systems is a challenge, which we now review.

1.3 Developing Interactive Programming Systems

Developing interactive programming systems requires the implementation of code editors and editor services for every programming language. Fortunately, IDEs are extensible, supporting reuse of its code editor and editor services by creating a *plugin* that connects the programming system of a programming language to the editor and editor services of the IDE [114]. Therefore, instead of developing an interactive programming system from scratch for each programming language, we use the code editor and editor services from IDEs.

However, developing responsive and correct interactive programming systems is a challenge, as IDEs do not provide support for implementing programming systems with these qualities, requiring manual application of the incrementality and scalability methods. Manually implementing incrementality is a challenge, as it requires the implementation of cross-cutting techniques such as dependency tracking, caching, cache invalidation, change detection, and persistence, which are complicated and error-prone to implement. Making

incrementality scale is even more challenging, as incrementality must scale up to tracking large dependency graphs, cache large amounts of data, do cache invalidation through these large graphs, and detect low-impact changes.

Finally, the programming system, and the incremental and scalable implementation of these parts, needs to be correct. However, since these methods are complicated and error-prone to implement, they cause subtle incrementality bugs that are hard to detect and reproduce, therefore reducing the correctness of the programming system. In summary, manual implementation of correct incremental and scalable interactive programming environments has a high development and maintenance effort, preventing us from developing interactive programming environments for the many DSLs that need to be developed for many problem domains.

What we need is a systematic approach to make interactive programming systems responsive and correct, without having to manually implement the complicated methods to achieve these qualities for every programming language. Therefore, we should use *language-parametric methods* for developing responsive and correct interactive programming systems. A language-parametric method takes as input an implementation or description of a programming language, and automatically produces an instance of that method, without the developer having to know much about the method at all. For example, a language-parametric incremental name and type analysis framework takes as input a language description, and automatically produces a correct and responsive name and type analysis, without the language developer having to worry about incrementality and scalability, and correctness thereof. This enables us to efficiently develop and maintain correct and responsive interactive programming environments for many DSLs.

The idea of language-parametric methods is not new. Therefore, we first describe several existing language-parametric methods, describe open problems, and then follow up with our contributions: language-parametric methods for developing responsive and correct interactive programming systems.

1.4 Language-Parametric Methods

One way of developing a programming language, is to use several disjunct but flexible *compiler-compiler* tools, which are tools that compile the compiler of a programming language. For example, one can specify a lexical analyzer (lexer) in tools such as Lex [19] or Flex [96], and then a context-free parser in Yacc [19] or Bison [96]. From such specifications, programs implementing a lexer and context-free analyzer are generated. A parser can then be created by feeding the tokens from the lexer into the context-free parser. Therefore, these tools are language-parametric methods for developing programming systems. However, these tools do not support the development of *interactive* programming systems, and are only available for a limited subset of the language development domain, such as parsing.

On the other hand, a *Language Workbench (LWB)* is a set of unified tools for developing (interactive) programming systems, with the goal of lowering the cost of developing and maintaining the programming system of DSLs. Al-

though language workbenches have been around since the 1980s, the term was coined by Martin Fowler only in 2005, in his blogpost "Language Workbenches: The Killer-App for Domain Specific Languages?" [46].

An example of an early language workbench is the Synthesizer Generator [123], which is a tool for generating editors (program synthesizers) from programming language descriptions, using incremental execution and inline error messages for responsive feedback. The ASF + SDF Meta-Environment [80, 14] language workbench included support for specifying program transformations and generation of interactive programming systems.

Modern language workbenches have been studied in the Language Workbench Challenge series of workshops, of which the 2013 edition resulted in a survey comparing language workbench features [39, 38]. For example, MetaEdit+ [78] is a platform-independent graphic language workbench for domain-specific modeling. MPS [159] is a projectional language workbench supporting non-textual notations such as tables and diagrams. Rascal [81] is a metaprogramming language and IDE for source code analysis and transformation. Finally, Spoofax [75] is a language workbench for specification of textual domain-specific languages with full IDE support.

Language workbenches provide language-parametric methods through *meta-languages*, which are programming languages that aid in the development of programming languages. Typically, meta-languages are domain-specific instead of general-purpose. For example, Spoofax provides SDF [153], a meta-DSL for syntax specification, Stratego [18], a meta-DSL for program analysis and transformation, and ESV [75], a meta-DSL for editor service specification. The SDF compiler generates a parse table and pretty-printer from an SDF syntax specification; the Stratego compiler generates an executable program analyzer and transformer; and the ESV specification is interpreted to provide editor services such as syntax highlighting and code completions. Therefore, Spoofax provides language-parametric methods for developing interactive programming systems.

There are three open problems with the language-parametric methods of language workbenches that we tackle in this dissertation: missing support for several sub-domains of language development, a lack of responsiveness, and a lack of integration between language-parametric methods.

Several subdomains of language development, such as name analysis and bootstrapping, have no domain-specific language-parametric methods, requiring these problems to be solved by manual encoding in a general-purpose method, reducing correctness and increasing the development and maintenance effort. We want to move to more domain-specific language-parametric methods, to benefit from the advantages of domain-specificity, such as direct expression of problems and solutions in terms of the domain, domain-specific consistency checking, and deriving multiple semantics from the same specification.

Furthermore, several language-parametric methods are not truly responsive; either requiring manual implementation of complicated methods such as scalable incrementality, reducing correctness and increasing effort; or cannot

be made responsive at all, reducing iteration times and increasing tedium. We want language-parametric methods for interactive programming environments to be responsive without much effort by the language developer.

Finally, there is a lack of integration between the language-parametric methods of language workbenches. Language workbenches have many components, such as multiple domain-specific meta-languages, their compilers or interpreters, generated artifacts of these compilers, compilers that may generate compilers, editor services, and so forth. All these components must integrate in a correct and responsive way, for the language workbench to be correct and responsive. However, this integration is typically manually implemented in an ad-hoc way, again increasing effort and reducing correctness and responsiveness. We need a systematic approach to integrate the language-parametric methods and components of language workbenches.

This dissertation addresses these open problems with five contributions.

1.5 Contributions

We now summarize the five core contributions of this dissertation.

1.5.1 NaBL: A Meta-DSL for Declarative Name Binding and Scope Rules

Every programming language needs to deal with names, their declarations and references, scopes, and importing of names into scopes. This is the name analysis step of the compiler of a programming system, which is often implemented manually with techniques such as nested environments maintained by tree traversals, or imperative lookup operations. However, this requires the language developer to think about *how* to develop name analysis, distracting from *what* the name binding rules of their language should be. Therefore, instead of manually implementing the name binding and scope rules of a language, we have developed a language-parametric method for *declaratively specifying* the name binding rules of a programming language with the Name Binding Language (NaBL), a domain-specific, declarative, meta-language.

With NaBL, a language developer declaratively specifies the name binding rules of their programming language in terms of definition and use sites of names, properties of these names associated to language constructs, namespaces for separating categories of names, scopes in which definitions are visible, and imports between scopes. From such a specification, we automatically derive a name analysis, and editor services for inline error checking, reference resolution, and code completion. We evaluate NaBL by specifying many name binding features of C# in NaBL, and by specifying common name binding patterns such as scoped, non-unique, globally visible, and subsequently visible definitions; and overloaded, type-directed, and nested references.

In conclusion, NaBL provides a language-parametric method for developing correct name analyses and corresponding editor services.

1.5.2 A Language Independent Task Engine for Incremental Name and Type Analysis

While NaBL provides a language-parametric method for name analysis, it does not support the definition of typing rules, and is not incremental, and therefore does not truly provide editor services for a responsive interactive programming environment. To mitigate this problem, we have developed a language independent task engine for incremental name and type analysis. In this approach, we specify naming rules in NaBL, and typing rules in TS – a meta-DSL for simple type system specification – from which we automatically derive a traversal that collects naming and typing tasks when given a program. Then, we collect tasks for a program and send them to the task engine, which incrementally executes changed tasks to incrementally execute name and type analysis, updating data structures required for editor services such as code completion, and responsively providing inline name and type error/warning messages.

We experimentally evaluate the correctness and responsiveness of our approach by running the incremental name and type analysis against the changes in the source code repository of a real-world application written in a real-world DSL. The evaluation shows that incremental and non-incremental analysis produce the same solution, showing correctness, and that for single-file changes, incremental analysis takes between 0.37 and 1.12 seconds, showing acceptable response times for interactive settings.

In conclusion, NaBL, TS, and the incremental task engine provide a language-parametric method for developing correct and responsive name and type analyses with corresponding editor services.

1.5.3 Bootstrapping Domain-Specific Meta-Languages of Language Workbenches

A *bootstrapped compiler* can compile its own source code, because the compiler is written in the compiled language itself. For example, the GCC compiler for the C language is a bootstrapped compiler; its source code is written in C and can compile itself. Bootstrapping yields several advantages:

- The compiler is written in the compiled high-level language.
- It provides a large-scale test case for detecting defects in the compiler and the compiled language.
- It shows that the language's coverage is sufficient to implement itself.
- Compiler improvements such as better static analysis or the generation of faster code applies to all compiled programs, including the compiler itself.

A *language workbench* provides high-level meta-languages (e.g. NaBL and TS) for developing DSLs and their compilers. Thus, users of a language workbench (language developers) develop their DSL in meta-languages, and therefore do not have to bootstrap their own language, which is good since DSLs have limited expressiveness and are often ill-suited for compiler development. What we desire instead, is *bootstrapping the meta-language compilers of language workbenches*, inheriting the benefits of bootstrapping stated above.

However, bootstrapping a language workbench is complicated by the fact

that most provide *multiple separate domain-specific meta-languages* for describing different language aspects such as syntax, name analysis, type analysis, code generation, and so forth. Thus, in order to build a meta-language compiler, multiple meta-language compilers need to be applied, entailing intricate dependencies that sound language workbench bootstrapping needs to handle. Furthermore, most language workbenches provide an interactive programming system for their meta-languages, supporting interactive development of DSLs. Therefore, bootstrapping operations must be available and observable in this interactive environment.

Our solution to these problems is to do versioning and dependency tracking between meta-languages, and perform *fixpoint bootstrapping*, where we iteratively self-apply meta-language compilers to derive new versions until no change occurs. Fixpoint bootstrapping is correct: it either produces a new baseline when it reaches a fixpoint, or stops and displays an error when it finds a defect (i.e., applying a meta-language compiler in an iteration failed), as long as meta-language compilers are deterministic and converge to a fixpoint. Furthermore, bootstrapping operations can be started, cancelled (when diverging), and rolled back (when defect) interactively, supporting the interactive programming system of the language workbench.

To evaluate our approach, we have implemented fixpoint bootstrapping for the Spoofax language workbench, and used it to successfully bootstrap eight meta-languages with seven changes. In conclusion, our approach provides a (meta)language-parametric method for correctly and interactively bootstrapping the meta-languages of language workbenches, in an interactive programming environment.

1.5.4 PIE: a DSL, API, and Runtime for Interactive Software Development Pipelines

A *software development pipeline* automates parts of the software engineering process, such as building software via build scripts, continuous integration testing and benchmarking on build farms, and automatic deployment of software artifacts to production servers. An *interactive software development pipeline* builds software artifacts, but also reacts immediately to changes in input, and provides timely feedback to the user. An interactive programming system is an instance of such a pipeline, where changes to programs are immediately processed to provide timely feedback to programmers.

However, interactivity complicates the development of pipelines, if responsiveness and correctness become the responsibility of the pipeline programmer, rather than being supported by the underlying system. Therefore, we need a system that is expressive enough to describe interactive software development pipelines, such as the interactive programming systems of LWBs, while still being correct and responsive.

Most build systems are not expressive enough, as they only support *static dependencies*, where all dependencies to files or build tasks have to be stated statically and up-front in the build script, whereas in a software development pipeline many dependencies only become evident *during build execution*. One build system that is expressive enough, is Pluto [36], a sound and optimal

incremental build system with support for *dynamic dependencies*, enabling build tasks to create dependencies to files and tasks during build execution, possibly based on (dynamic) values produced by previous build tasks. However, Pluto suffers from four open problems affecting ease of development: requiring a lot of Java boilerplate to define build tasks; semi-automated persistence, requiring pipeline programmers to manually reason about where to cache task outputs; explicit dependency tracking where dependencies could be inferred; and missing domain-specific software development pipeline features such as file system paths and support for lists.

To solve these problems, we have developed Pipelines for Interactive Environments (PIE), a DSL, Application Program Interface (API), and runtime for developing correct and responsive interactive software development pipelines, where ease of development *is* a focus. The PIE DSL serves as a front-end for developing pipelines with minimal boilerplate in a functional language with support for concepts from the interactive software development pipeline domain such as dependencies, filesystem paths and operations, and list operations. The PIE API is a lower-level front-end for developing foreign pipeline functions which cannot be modeled in the DSL, while having reduced boilerplate compared to Pluto, and also serves as a compilation target for the DSL. Finally, the runtime incrementally executes pipelines implemented in the API using Pluto's incremental build algorithm, while fully automating persistence and inferring dependencies where possible.

We evaluate PIE with two case studies, one being the reimplementation of a significant part of the interactive programming system of the Spoofax language workbench in PIE. The existing pipeline of Spoofax's interactive programming system was scattered across four different formalisms, decreasing ease of development; overapproximates dependencies, causing loss of incrementality; and underapproximates dependencies, causing loss of correctness. However, with PIE, we can easily integrate the different components of Spoofax; such as its parser, analyzers, transformations, build scripts, editor services, meta-languages, and dynamic language loading; into a single formalism. PIE ensures that the pipeline is correct and responsive, without the pipeline programmer having to implement techniques such as incrementality, or without having to reason about correctness. In conclusion, PIE provides a language-parametric method for developing interactive software development pipelines, a superset of correct and responsive interactive programming environments.

1.5.5 Scalable Incremental Building with Dynamic Task Dependencies

Previous work on PIE builds forth on Pluto by improving its ease of use, but essentially uses the same incremental build algorithm. To make a build up-to-date after changes, the Pluto incremental build algorithm traverses the entire dependency graph (produced in a previous build) from top to bottom, while re-executing tasks affected by a change, and possibly executing new tasks. This enables Pluto to detect changes to files and tasks without the user having to tell Pluto what has actually changed, while also elegantly discovering changes to dynamic dependencies by (re-)executing (new) tasks. However, the downside is

that this algorithm does not scale, because the traversal is dependent on the size of the dependency graph, *not* the impact of the change. This quickly becomes a problem in interactive programming systems, where there are many changes and those changes have a low-impact (e.g., programmer typing characters into an editor), while the program and its induced dependency graph is large. For example, in the Spoofax language workbench pipeline, we observed ~ 3 second build times even when nothing has changed. Therefore, we need a new incremental build algorithm that *scales down* to many low-impact changes, while *scaling up* to large dependency graphs, while still supporting dynamic dependencies.

To solve this scalability problem, we have developed a new incremental build algorithm that performs change-driven rebuilding. It takes as input a set of changed files, starts rebuilding directly affected tasks from the changed leaves of the dependency graph, and rebuilds transitively affected tasks, while also accounting for new task dependencies discovered during rebuilding. Our algorithm scales with the impact of a change, and is independent from the size of the dependency graph, because it only ever visits affected tasks.

We experimentally evaluate our change-driven bottom-up algorithm by comparison against Pluto's top-down algorithm. As a subject, we use the Spoofax-PIE pipeline, a real-world build script for the interactive programming system of the Spoofax language workbench. To measure incremental performance an scalability, we synthesized a chain of 60 realistic changes of varying types and impacts. Results show that for low-impact changes (i.e., changes that only cause a small number of tasks to be actually affected), our change-driven algorithm is several orders of magnitude faster than Pluto's top-down algorithm, while not slower for high-impact changes.

In conclusion, our new algorithm makes PIE scalable, in addition to being correct and responsive. This in turn makes the Spoofax PIE pipeline scalable, providing a language-parametric method for developing truly correct and responsive interactive programming systems.

1.6 Research Methodology

We now describe the research methodology used in the core contributions of this dissertation.

Mary Shaw identified five types of research questions [128] based on the submissions to the International Conference on Software Engineering (ICSE). The type of question we answer in this dissertation is a "method or means of development". That is, what is a better way to develop (or: how do we automate the development of) correct and responsive interactive programming systems? Answering this question requires *designing* new (language-parametric) methods for developing interactive programming systems. In order to design these methods systematically, we follow the iterative approach of the memorandum on design-oriented information systems research [113], consisting of four phases: analysis, design, evaluation, and diffusion.

In the analysis phase, we identify and describe open problems in the development of interactive programming systems, and analyze the relation to existing

approaches. In the design phase, we design new methods for developing interactive programming systems that solve open problems, and create tools that implement those methods. These tools come in the form of meta-DSLs and their compilers, APIs that can be programmed against, algorithms that execute instances of a generic model, and systems that combine these artifacts.

In the evaluation phase, we evaluate to what degree our methods and tools solve open problems. When the open problem is an ease of development problem, we evaluate coverage of our method by application to real-world case studies, show this application in a research paper or external artifact, and discuss to what degree we improve ease of development. When the open problem is a performance problem, we experimentally evaluate our approach against real-world case studies and subjects, measure performance differences, show the measurement results in a research paper, and discuss to what degree we improve performance. We experimentally evaluate the correctness of our approaches with the same case studies and subjects.

In the diffusion phase, we publish our findings as research papers to conferences or journals and present our research at these conferences. Furthermore, we try to apply our research in industry to gain further insights into applicability and what can be improved, and apply our research to our own systems (dogfooding), possibility spinning up new research projects.

1.7 Structure

We now discuss the rest of the structure of this dissertation. The main chapters of this dissertation are based on five peer-reviewed publications. The author of this dissertation is the main contributor of all publications, and the first author for four of the publications. In the publication for chapter 3, Guido Wachsmuth is the first author, since he did most of the writing, whereas the author of this dissertation implemented the task engine, implemented the benchmark subjects, helped with benchmarking, and wrote parts of the introduction and evaluation sections, and the related work section.

Since each main chapter is based on a stand-alone publication with distinct contributions, there is some redundancy, especially in the introduction sections. However, we chose not to remove that redundancy, in order to ensure that each chapter can be read independently. The main chapters and their corresponding publication are as follows:

- Chapter 2 is an updated version of the SLE 2012 paper *Declarative Name Binding and Scope Rules* [89].
- Chapter 3 is an updated version of the SLE 2013 paper *A Language Independent Task Engine for Incremental Name and Type Analysis* [160].
- Chapter 5 is an updated version of the GPCE 2016 paper *Bootstrapping Domain-Specific Meta-Languages in Language Workbenches* [86].
- Chapter 7 is an updated version of the Programming 2018 paper *PIE: A Domain-Specific Language for Interactive Software Development Pipelines* [91].
- Chapter 8 is an updated version of the ASE 2018 paper *Scalable Incremental*

Building with Dynamic Task Dependencies [87].

We reflect on incremental name and type analysis, bootstrapping, and our engineering work on Spoofax Core in chapter 4, and reflect on language workbench pipelines in chapter 6. Finally, we end with a conclusion in chapter 9 where we summarize our work, discuss the work in relation to the thesis, and discuss future work.

NaBL: A Meta-DSL for Declarative Name Binding and Scope Rules

2

Abstract

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

2.1 Introduction

Software language engineering is concerned with *linguistic abstraction*, the formalization of our understanding of domains of computation in higher-level software languages. Such languages allow direct expression in terms of the domain, instead of requiring encoding in a less specific language. They raise the level of abstraction and reduce accidental complexity.

One of the key goals in the field of language engineering is to apply these techniques to the discipline itself: high-level languages to specify all aspects of software languages. Declarative languages are of particular interest since they enable language engineers to focus on the *What?* instead of the *How?*.

Syntax definitions are a prominent example. With declarative formalisms such as Extended Backus–Naur Form (EBNF), we can specify the syntactic concepts of a language without specifying how they can be recognized programmatically. This declarativity is crucial for language engineering. Losing it hampers evolution, maintainability, and compositionality of syntax definitions [76].

Despite the success of declarative syntax formalisms, we tend to programmatic specifications for other language aspects. Instead of specifying languages, we build programmatic language processors, following implementation patterns in rather general specification languages. These languages might still be considered domain-specific, when they provide special means for programmatic language processors. They also might be considered declarative, when they abstract over computation order. However, they enable us only to implement language processors faster, but not to specify language aspects. They lack

domain concepts for these aspects and focus on the *How?*. That is a problem since (1) it entails overhead in encoding concepts in a programming language and (2) the encoding obscures the intention; understanding the definition requires decoding.

Our goal is to extend the set of really declarative, domain-specific languages for language specifications. In this paper, we are specifically concerned with *name binding and scope rules*. Name binding is concerned with the relation between definitions and references of identifiers in textual software languages, including scope rules that govern these relations. In language processors, it is crucial to make information about definitions available at the references. Therefore, traditional language processing approaches provide programmatic abstractions for name binding. These abstractions are centered around tree traversal and information propagation from definitions to references. Typically, they are not specifically addressing name binding, but can also be used for other language processing tasks such as compilation and interpretation.

Name binding plays a role in multiple language engineering processes, including editor services such as reference resolution, code completion, refactorings, type checking, and compilation. The different processes need different information about definitions. For example, name resolution tries to find one definition, while code completion needs to determine all possible references in a certain place. The different requirements lead either to multiple re-implementations of name binding rules for each of these purposes, or to non-trivial, manual weaving into a single implementation supporting all purposes. This results in code duplication with as result errors, inconsistencies, and increased maintenance effort.

The traditional paradigm influences not only language processing, but also language specification. For example, the Object Constraint Language (OCL) standard [111] specifies name binding in terms of nested environments, which are maintained in a tree traversal. The C# language specification [134] defines name resolution as a sequence of imperative lookup operations. In this paper, we abstract from the programmatic mechanics of name resolution. Instead, we aim to declare the roles of language constructs in name binding and leave the resolution mechanics to a generator and run-time engine. We introduce the *NaBL*, a language with linguistic abstractions for declarative definition of name binding and scope rules. NaBL supports the declaration of definition and use sites of names, properties of these names associated with language constructs, namespaces for separating categories of names, scopes in which definitions are visible, and imports between scopes.

NaBL is integrated in the Spoofax Language Workbench [75], but can be reused in other language processing environments. From definitions in the name binding language, a compiler generates a language-specific name resolution strategy in the Stratego rewriting language [151] by parametrizing an underlying generic, language independent strategy. Name resolution results in a persistent symbol table for use by semantic editor services such as reference resolution, consistency checking of definitions, type checking, refactoring, and code generation. The implementation supports multiple file analysis by

default.

We proceed as follows. In sections 2.2 and 2.3 we introduce NaBL by example, using a subset of the C# language. In section 2.4 we discuss the derivation of editor services from a name binding specification. In section 2.5 we give a high-level description of the generic name resolution algorithm underlying NaBL. In section 2.6 we discuss the integration of NaBL into the Spoofax Language Workbench. In section 2.7 we discuss NaBL's applicability to different languages. Finally, section 2.8 discusses related work.

2.2 Declarative Name Binding and Scope Rules

In this section we introduce NaBL, illustrated with examples drawn from the specification of name binding for a subset of C# [134]. Listing 2.1 defines the syntax of the subset in Syntax Definition Formalism (SDF) [153]. The subset is by no means complete; it has been selected to model representative features of name binding rules in programming and domain-specific languages. In the following subsections we discuss the following fundamental concepts of name binding: *definition and use sites*, *namespaces*, *scopes*, and *imports*. For each concept we give a general definition, illustrate it with an example in C#, and then we show how the concept can be modeled in NaBL.

2.2.1 Definitions and References

The essence of name binding is establishing relations between a *definition* that *binds* a name and a *reference* that *uses* that name. Name binding is typically defined programmatically through a *name resolution algorithm* that connects references to definitions. A *definition site* is the location of a definition in a program.

In many cases, definition sites are required to be *unique*, that is, there should be exactly one definition site for each name. However, there are cases where definition sites are allowed to be *non-unique*.

Example. Listing 2.2a contains class definitions in C#. Each class definition binds the name of a class. Thus, we have definition sites for `A`, `B`, and `C`. Base class specifications are references to these definition sites. In the example, we have references to `A` as the base class of `B` and `B` as the base class of `C`. (Thus, `B` is a sub-class of, or inherits from `A`.) There is no reference to `C`.

The definition sites for `A` and `B` are unique. By contrast, there are two definition sites for `C`, defining parts of the same class `C`. Thus, these definition sites are non-unique. This is correct in C#, since regular class definitions are required to be unique, while partial class definitions are allowed to be non-unique.

Abstract Syntax Terms. In Spoofax, abstract syntax trees (ASTs) are represented using first-order terms. Terms consist of strings (`"x"`), lists of terms (`["x","y"]`), and constructor applications (`ClassType("A")`) for labelled tree nodes with a fixed number of children. Annotations in grammar productions (listing 2.1) define the constructors to be used in AST construction. For example, `Class(Partial(), "C", Base("B"), [])` is the representation of the

```
Using* NsMem*                                  -> CompUnit     {"Unit"}

"using" NsOrTypeName ";"                       -> Using        {"Using"}
"using" ID "=" NsOrTypeName                    -> Using        {"Alias"}
ID                                             -> NsOrTypeName {"NsOrType"}
NsOrTypeName "." ID                            -> NsOrTypeName {"NsOrType"}

"namespace" ID "{" Using* NsMem* "}"           -> NsMem        {"Namespace"}
Partial "class" ID Base "{" ClassMem* "}"      -> NsMem        {"Class"}

                                               -> Partial      {"NonPartial"}
"partial"                                      -> Partial      {"Partial"}
                                               -> Base         {"NoBase"}
":" ID                                         -> Base         {"Base"}

Type ID ";"                                    -> ClassMem     {"Field"}
RetType ID "(" {Param ","}* ")" Block ";"      -> ClassMem     {"Method"}

ID                                             -> Type         {"ClassType"}
"int"                                          -> Type         {"IntType"}
"bool"                                         -> Type         {"BoolType"}
Type                                           -> RetType
"void"                                         -> RetType      {"Void"}
Type ID                                        -> Param        {"Param"}

"{" Stmt* "}"                                  -> Block        {"Block"}
Decl                                           -> Stmt
EmbStmt                                        -> Stmt
"return" Exp ";"                               -> Stmt         {"Return"}
Type ID ";"                                    -> Decl         {"Var"}
Type ID "=" Exp ";"                            -> Decl         {"Var"}
Block                                          -> EmbStmt
StmtExp ";"                                    -> EmbStmt
"foreach" "(" Type ID "in" Exp ")" EmbStmt     -> EmbStmt      {"Foreach"}

INT                                            -> Exp          {"IntLit"}
"true"                                         -> Exp          {"True"}
"false"                                        -> Exp          {"False"}
ID                                             -> Exp          {"VarRef"}
StmtExp                                        -> Exp
Exp "." ID                                     -> StmtExp      {"FieldAccess"}
Exp "." ID "(" {Exp ","}* ")"                  -> StmtExp      {"Call"}
ID "(" {Exp ","}* ")"                          -> StmtExp      {"Call"}
```

Listing 2.1: Syntax definition in SDF2 for a subset of C#. The names in the annotations are abstract syntax tree constructors.

```
class A {}
class B:A {}
partial class C:B{
}
partial class C {}
```

(a) Class declarations in C#.

```
rules
  Class(NonPartial(), c, _, _): defines unique class c
  Class(Partial(), c, _, _)   : defines non-unique class c
  Base(c)                     : refers to class c
  ClassType(c)                : refers to class c
```

(b) NaBL specification of definitions and references for C# class names.

Listing 2.2: Definitions and references of names.

```
class x {
  int x;

  void x() {
    int x;
    x = x + 1;
  }

}
```

(a) Homonym declarations in C#.

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

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

(b) NaBL specification for different C# namespaces.

Listing 2.3: Namespaces of definitions and use sites.

first partial class in listing 2.2a. A term *pattern* is a term that may contain variables (`x`) and wildcards (`_`).

Model. A specification in NaBL consists of a collection of rules of the form `pattern : clause*`, where `pattern` is a term pattern and `clause*` is a list of name binding declarations about the language construct that matches with `pattern`.

Listing 2.2b shows a declaration of the definitions and references for class names in C#. The first two rules declare class definition sites for class names. Their patterns distinguish regular (non-partial) and partial class declarations. While non-partial class declarations are unique definition sites, partial class declarations are non-unique definition sites. The third rule declares that the term pattern `Base(c)` is a reference to a class with name `c`. Thus, the "`: A`" in listing 2.2a is a reference to class `A`. Similarly, the second rule declares a class type as a reference to a class.

2.2.2 *Namespaces*

Definitions and references declare relations between named program elements and their uses. Languages typically distinguish several *namespaces*, i.e. different kinds of names, such that an occurrence of a name in one namespace is not related to an occurrence of that same name in another.

Example. Listing 2.3a shows several definitions for the same name `x`, but of different kinds, namely a class, a field, a method, and a variable. Each of these kinds has its own namespace in C#, and each of these namespaces has its own

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

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

(a) Scoped homonym method and variable declarations in C#.

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

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

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

  Block(_): scopes variable
```

(b) NaBL specification of scopes for different C# namespaces.

Listing 2.4: Named and anonymous lexical scopes.

name `x`. This enables us to distinguish the definition sites of class `x`, field `x`, method `x`, and variable `x`, which are all unique.

Model. We have declared definitions and references for the namespace `class` already in the previous example. Listing 2.3b extends that declaration covering also the namespaces `field`, `method`, and `variable`. Note that it is required to declare namespaces to ensure the consistency of name binding rules. Definition sites are bound to a single namespace (`defines class c`), but use sites are not. For example, a variable in an expression might either refer to a variable, or to a field, which is modeled in the last rule. In our example, this means that variable declarations hide field declarations, because variables are resolved to variables, if possible. Thus, both `x` in the assignment in listing 2.3a refer to the variable `x`.

2.2.3 Scopes

Scopes restrict the visibility of definition sites. A *named scope* is the definition site for a name which scopes other definition sites. By contrast, an *anonymous scope* does not define a name. Scopes can be nested and name resolution typically looks for definition sites from inner to outer scopes.

Example. Listing 2.4a includes two definition sites for a method `m`. These definition sites are not distinguishable by their namespace `method` and their name `m`, but, they are distinguishable by the scope they are in. The first definition site resides in class `C`, the second one in class `D`. In C#, class declarations scope method declarations. They introduce named scopes, because class declarations are definition sites for class names. The listing also contains three definition sites for a variable `x`. Again, these are distinguishable by their scope.

In C#, method declarations and blocks scope variable declarations. Method declarations are named scopes, blocks are anonymous scopes. The first definition site resides in method `m` in class `C`, the second one in method `m` in class

```
namespace N {
  class N {}

  namespace N { class N {} }
}
```

(a) Nested namespace declarations in C#.

```
namespaces namespace
rules
  Namespace(n, _):
    defines namespace n
    scopes namespace, class
```

(b) NaBL specification for C# nested namespace declarations.

Listing 2.5: Nested C# namespaces.

`D`, and the last one in a nameless block inside method `m` in class `D`. In the assignment inside the block (line 9), `x` refers to the variable declaration in the same block, while the `x` in the outer assignment (line 10) refers to the variable declaration outside the block. In both assignments, `y` refers to the variable declaration in the outer scope, because the block does not contain a definition site for `y`.

Model. The `scopes ns` clause in NaBL declares a construct to be a scope for namespace `ns`. Listing 2.4b declares scopes for fields, methods, and variables. Named scopes are declared at definition sites. Anonymous scopes are declared similarly, but lack a `defines` clause.

2.2.4 Namespaces as Language Concepts

C# has a notion of 'namespaces'. It is important to distinguish these *namespaces as a language concept* from *namespaces as a naming concept*, which group names of different kinds of declarations. Specifically, in C#, namespace declarations are top-level scopes for class declarations. Namespace declarations can be nested.

Example. Listing 2.5a declares a top-level namespace `N`, scoping a class declaration `N` and an inner namespace declaration `N`. The inner namespace declaration scopes another class declaration `N`. The definition sites of the namespace name `N` and the class name `N` are distinguishable, because they belong to different namespaces (as a naming concept). The two definition sites of namespace name `N` are distinguishable by scope. The outer namespace declaration scopes the inner one. Also, the definition sites of the class name `N` are distinguishable by scope. The first one is scoped by the outer namespace declaration, while the second one is scoped by both namespace declarations.

Model. The names of C# namespace declarations are distinguishable from names of classes, fields, etc. As declared in listing 2.5b, their names belong to the `namespace` namespace. The name binding rules for definition sites of names of this namespace models the scoping nature of C# namespace declarations.

2.2.5 Imports

An import introduces into the current scope definitions from another scope, either under the same name or under a new name. An import that imports all definitions can be transitive.

```
using N;

namespace M {
  class C {
    int f;
  }
}

namespace O {
  using D = M.C;
  class E:D {
    void m() {}
  }
  class F:E { }
}
```

(a) Various forms of imports in C#.

```
rules
  Using(qname):
    imports class from namespace ns
    where qname refers to namespace ns

  Alias(alias, qname):
    imports namespace ns as alias
    where qname refers to namespace ns
    otherwise imports class c as alias
    where qname refers to class c

  Base(c):
    imports field (transitive),
            method (transitive)
            from class c
```

(b) NaBL specification of C# import mechanisms.

Listing 2.6: Importing definitions from another scope.

Example. Listing 2.6a shows different kinds of imports in C#. First, a `using` directive imports type declarations from namespace `N`. Second, another `using` directive imports class `C` from namespace `M` into namespace `O` under a new name `D`. Finally, classes `E` and `F` import fields and methods from their base classes. These imports are transitive, that is, `F` imports fields and methods from `E` and `D`.

Model. Listing 2.6b shows name binding rules for import mechanisms in C#. The first rule handles `using` declarations, which import all classes from the namespace to which the qualified name `qname` resolves to. The second rule models aliases, which either import a namespace or a class under a new name, depending on the resolution of `qname`. The last rule models inheritance, where fields and methods are imported transitively from the base classes into the current (parent class) scope.

2.2.6 *Types*

So far, we discussed names, namespaces, and scopes to distinguish definition sites for the same name. Types also play a role in name resolution and can be used to distinguish definition sites for a name or to find corresponding definition sites for a use site.

Example. Listing 2.7a shows a number of overloaded method declarations. These share the same name `m`, namespace `method`, and scope class `C`. But we can distinguish them by the types of their parameters. Furthermore, all method calls inside method `x` can be uniquely resolved to one of these methods by taking the argument types of the calls into account.

Model. Listing 2.7b includes type information into name binding rules for fields, methods, and variables. Definition sites might have types. In the simplest case, the type is part of the declaration. In the example, this holds for parameters. For method calls, the type of the definition site for a method name

```
class C {
  void m() {}
  void m(int x) {}
  void m(bool x) {}
  void m(int x, int y) {}
  void m(bool x, bool y) {}
  void x() {
    m();
    m(42);
    m(true);
    m(21, 21);
    m(true, false);
  }
}
```

(a) Overloaded method declarations in C#.

```
rules
  Method(t, m, p*, _):
    defines unique method m
      of type (t*, t)
    where p* has type t*

  Call(m, a*):
    refers to method m
      of type (t*, _)
    where a* has type t*

  Param(t, p):
    defines unique variable p
      of type t
```

(b) Types in name binding rules for C# overloaded methods.

Listing 2.7: Types of definitions, and type-dependent use sites.

depends on the types of the parameters. A type system is needed to connect the type of a single parameter, as declared in the rule for parameters, and the type of a list of parameters, as required in the rule for methods. We will discuss the influence of a type system and the interaction between name and type analysis later. For now, we assume that the type of a list of parameters is a list of types of these parameters.

Type information is also needed to resolve method calls to possibly overloaded methods. The `refers` clause for method calls therefore requires the corresponding definition site to match the type of the arguments. Again, we omit the details how this type can be determined. We also do not consider subtyping here. Method calls and corresponding method declarations need to have the same argument and parameter types.

2.3 Name Binding Patterns

We now identify typical name binding patterns. These patterns are formed by scopes, definition sites and their visibility, and use sites referencing these definition sites. We explain each pattern first and give an example in C# next. Afterwards, we show how the example can be modeled with declarative name binding rules.

2.3.1 Unscoped Definition Sites

In the simplest case, definition sites are not scoped and globally visible.

Example. In C#, namespace and class declarations (as well as any other type declaration) can be unscoped. They are globally visible across file boundaries. For example, the classes `C1`, `C2`, and `C3` in Listing 2.2a are globally visible. In listing 2.3a, only the outer namespace `N` is globally visible.

In contrast to C#, C++ has file scopes and all top-level declarations are only visible in a file. To share global declarations, each file has to repeat the

```
rules
  CompilationUnit(_, _):
    scopes namespace, class

  (f, CompilationUnit(_, _)):
    defines file f
    scopes namespace, class
```

Listing 2.8: Two ways to model file scope for top-level syntax tree nodes in NaBL.

```
rules
  Namespace(n, _):
    defines non-unique namespace n in surrounding scope

  Var(t, c):
    defines unique variable of type t in subsequent scope
```

Listing 2.9: NaBL specification of the visibility of definition sites inside scopes.

declaration and mark it as `extern`. This is typically achieved by importing a shared header file.

Model. We consider any definition site that is not scoped by another definition site or by an anonymous scope to be in global scope. These definition sites are visible over file boundaries. File scope can be modeled with a scoping rule in two different ways. Both are illustrated in listing 2.8.

The first rule declares the top-level node of abstract syntax trees as a scope for all namespaces which can have top-level declarations. This scope will be anonymous, because the top-level node cannot be a definition site (otherwise this definition site would be globally visible). The second rule declares a tuple consisting of file name and the abstract syntax tree as a scope. This tuple will be considered a definition site for the file name. Thus, the scope will be named after the file.

2.3.2 *Definition Sites inside their Scopes*

Typically, definition sites reside inside the scopes where they are visible. Such definition sites can either be visible only after their declaration, or everywhere in their surrounding scope.

Example. In C#, namespace members such as nested namespace declarations and class declarations are visible in their surrounding scope. The same holds for class members. In contrast, variable declarations inside a method scope become visible only after their declaration.

Model. Scoped definition sites are by default visible in the complete scope. Optionally, this can be stated explicitly in `defines` clauses. Listing 2.9 illustrates this for namespace declarations. The second rule in this listing shows how to model definition sites which become visible only after their declaration.

```
class C {
  void m(int[] x) {
    foreach(int x in x)
      System.Console.WriteLine(x);
  }
}
```

(a) Loop with scoped iterator variable `x` in C#.

```
rules

  Foreach(t, v, exp, body):
    defines unique variable v
      of type t
      in body
```

(b) NaBL specification of C# foreach loops.

Listing 2.10: Definition sites outside of their scopes.

```
rules
  Seq(Var(t, v), stmts): defines unique variable v of type t in stmts

  [Var(t, v) | stmts]  : defines unique variable v of type t in stmts
```

Listing 2.11: Two ways to model definition sites becoming visible after their declaration.

2.3.3 *Definition Sites outside their Scopes*

Some declarations include not only the definition site for a name, but also the scope for this definition site. In such declarations, the definition site resides outside its scope.

Example. Let expressions are a classical example for definition sites outside their scopes. In C#, `foreach` statements declare iterator variables, which are visible in embedded statements. Listing 2.10a shows a method with a parameter `x`, followed by a `foreach` statement with an iterator variable of the same name. This is considered incorrect in C#, because definition sites for variable names in inner scopes collide with definition sites of the same name in outer scopes. However, the use sites can still be resolved based on the scopes of the definition sites. The use site for `x` inside the loop refers to the iterator variable, while the `x` in the collection expression refers to the parameter.

Model. Listing 2.10b shows the name binding rule for `foreach` loops, stating the scope of the variable explicitly. Note that definition sites which become visible after their declaration are a special case of this pattern. Listing 2.11 illustrates how this can be modeled in the same way as the `foreach` loop. The first rule assumes a nested representation of statement sequences, while the second rule assumes a list of statements.

2.3.4 *Contextual Use Sites*

Definition sites can be referenced by use sites outside of their scopes. These use sites appear in a context which determines the scope into which they refer. This context can either be a direct reference to this scope, or has a type which determines the scope.

Example. In C#, namespace members can be imported into other namespaces. Listing 2.5a shows a class `N` in a nested namespace. In listing 2.12a, this class is

```
using N.N.N;


namespace N' {
  class C {
    C f;
    void m(C p) { }
  }
  class D {
    void m(C p) {
      p.m(p.f);
    }
  }
}
```

(a) Contextual use sites in C#.

```
rules
  NsOrType(n1, n2):
    refers to namespace n2 in ns
    otherwise to class n2 in ns
    where n1 refers to namespace ns

  FieldAccess(e, f):
    refers to field f in c
    where e has type ClassType(c)

  MethodCall(e, m, p*):
    refers to method m of type (t*, _) in c
    where e has type ClassType(c)
    where p* has type t*
```

(b) NaBL specification of contextual use sites.

Listing 2.12: Contextual use sites.

imported. The `using` directive refers to the class with a qualified name. The first part of this name refers to the outer namespace `N`. It is the context of the second part, which refers to the inner namespace `N`. The second part is then the context for the last part of the qualified name, which refers to the class `N` inside the inner namespace.

Listing 2.12a also illustrates use sites in a type-based context. In method `m` in class `D`, a field `f` is accessed. The corresponding definition site is outside the scope of the method in class `C`. But this scope is given by the type of `p`, which is the context for the field access. Similarly, the method call is resolved to method `m` in class `C` because of the type of `p`.

Model. Listing 2.12b illustrates how to model contextual use sites. The scope of the declaration site corresponding to a use site can be modeled in `refers` clauses. This scope needs to be determined from the context of the use site. The first rule resolves the context of a qualified name part to a namespace `ns` and declares the use site to refer either to a namespace or to a class in `ns`. The remaining rules declare use sites for field access and method calls. They determine the type of the context, which needs to be a class type. A field access refers to a field in that class. Similarly, a method call refers to a method with the right parameter types in that class.

2.4 Editor Services

Modern IDEs provide a wide range of editor services where name resolution plays a large role. Traditionally, each of these services would be handcrafted for each language supported by the IDE, requiring substantial effort. However, by accurately modeling the relations between names in NaBL, it is possible to generate a name resolution algorithm and editor services that are based on that algorithm.

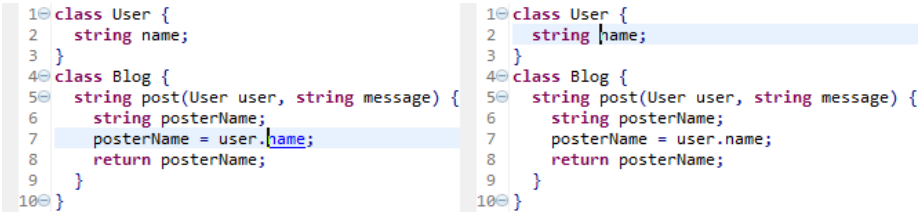


Figure 2.1: Reference resolution of `name` field reference to `name` field definition, derived from a NaBL specification

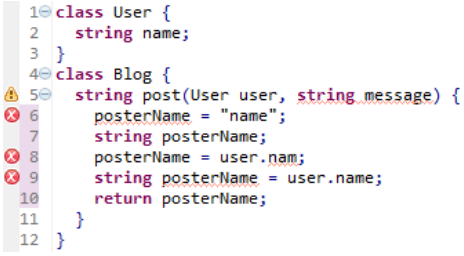


Figure 2.2: Error checking derived from NaBL rules. `posterName` is declared twice, and `nam` is not declared at all.

2.4.1 *Reference Resolving*

Name resolution is exposed directly in the IDE in the form of reference resolving: press and hold Control and hover the mouse cursor over an identifier to reveal a blue hyperlink that leads to its definition side. This behavior is illustrated in fig. 2.1.

2.4.2 *Constraint Checking*

Modern IDEs statically check programs against a wide range of constraints. Constraint checking is done on the fly while typing and directly displayed in the editor via error markers on the text and in the outline view. Error checking constraints are generated from the NaBL for common name binding errors such as unresolved references, duplicate definitions, use before definition and unused definitions.

Figure 2.2 shows an editor with error markers. The `message` parameter in the `post` method has a warning marker indicating that it is not used in the method body. On the line that follows it, the `posterName` variable is assigned but has not yet been declared, violating the visibility rules of listing 2.9. Other errors in the method include a subsequent duplicate definition of `posterName`, which violates the uniqueness constraint of the `variable` namespace of listing 2.3b, and referencing a non-existent property `nam`.

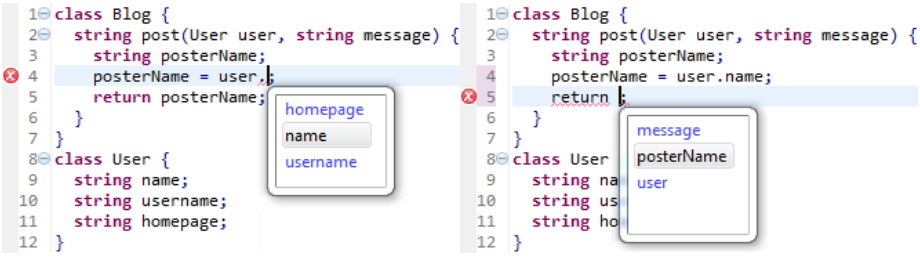


Figure 2.3: Code completion for fields and local variables.

2.4.3 *Code Completion*

With code completion, partial (or empty) identifiers can be completed to full identifiers that are valid at the context where code completion is executed. Figure 2.3 shows an example of code completion. In the left program code completion is triggered on a field access expression on the `user` object. The `user` object is of type `User`, so all fields of `User` are shown as candidates. On the right, completion is triggered on a variable reference, so all variables in the current scope are shown.

2.5 Implementation

To implement name resolution based on NaBL, we employ a name resolution algorithm that relies on a symbol table data structure to persist name bindings and lazy evaluation to resolve all references. In this section we give an overview of the data structure, the name resolution algorithm, and their implementation.

2.5.1 *Persistence of Name Bindings*

To persist name bindings, each definition and reference is assigned a qualified name in the form of a Uniform Resource Identifier (URI). The URI identifies the occurrence across a project. Use sites share the URIs of their corresponding definition sites.

A URI consists of the namespace, the path, and the name of a definition site. As an example, the URI `method://N/C/m` is assigned to a method `m` in a class `C` in a namespace `N`. Here, the segments represent the names of the scopes. Anonymous scopes are represented by a special path segment `anon(u)`, where `u` is a unique string to distinguish different anonymous scopes. For use in analyses and transformations, URIs can be represented in the form of Abstract Syntax Trees (ASTs). For example, `[method(),"N","C","m"]` is the URI for method `m`.

All name bindings are persisted in an in-memory data structure called the semantic index. It consists of a symbol table that lists all URIs that exist in a project, and can be efficiently implemented as a hash table. It maps each URI to the file and offset of their occurrences in the project. It can also store additional information, such as the type of a definition.

2.5.2 Resolving Names

Our algorithm is divided into three phases. First, in the annotation phase, all definition and use sites are assigned a preliminary URI, and definition sites are stored in the index. Second, definition sites are analyzed, and their types are stored in the index. And third, any unresolved references are resolved and stored in the index.

Annotation Phase. In the first phase, the AST of the input file is traversed in top-down order. The logical nesting hierarchy of programs follows from the AST, and is used to assign URIs to definition sites. For example, as the traversal enters the outer namespace scope `n`, any definitions inside it are assigned a URI that starts with '`n.`'. As a result of the annotation phase, all definition and use sites are annotated with a URI. In the case of definition sites, this is the definitive URI that identifies the definition across the project. For references, a temporary URI is assigned that indicates its context, but the actual definition it points to has to be resolved in a following phase. For reference by the following phases, all definitions are also stored in the index.

Definition Site Analysis Phase. The second phase analyzes each definition site in another top-down traversal. It determines any local information about the definition, such as its type, and stores it in the index so it can be referenced elsewhere. Types and other information that cannot be determined locally are determined and stored in the index in the last phase.

Use Site Analysis Phase. When the last phase commences, all local information about definitions has been stored in the index, and non-local information about definitions and uses in other files is available. What remains is to resolve references and to determine types that depend on non-local information (in particular, inferred types). While providing a full description of the use site analysis phase and the implementation of all name binding constructs is outside the scope of this paper, the below steps sketch how each reference is resolved:

1. Determine the temporary URI `ns://path/n` which was annotated in the first analysis phase.
2. If an import exists in scope, expand the current URI for that import.
3. If the reference corresponds to a name-binding rule that depends on non-local information such as types, retrieve that information.
4. Look for a definition in the index with namespace `ns`, path `path`, and name `n`. If it does not exist, try again with a prefix of `path` that is one segment shorter. If the no definition is found this way, store an error for the reference.
5. If the definition is an alias, resolve it.

An important part to highlight in the algorithm is the interaction between name and type analysis that happens for example with the `FieldAccess` expression of listing 2.12b. For name binding rules that depend on types or other non-local information, it is possible that determining the type recursively trig-

gers name resolution. For this reason, we apply lazy evaluation, ensuring that any reference can be resolved lazily as requested in this phase. By traversing through the entire tree, we ensure that all use sites are eventually resolved and persisted to the index.

2.6 Integration into Spoofax

NaBL, together with the index, is integrated into the Spoofax Language Workbench. Stratego rules are generated by the NaBL compiler that use the index API to interface with Spoofax. In this section we will show the index API and how the API is used to integrate the editor services seen in section 2.4.

2.6.1 Index API

Once all analysis phases have been completed, the index is filled with a summary of every file. To use the summaries we provide the index API with a number of lookups and queries. Lookups transform annotated identifiers into definitions. Queries transform definitions (retrieved using a lookup) into other data. The API is used for integrating editor services, but is also exposed to Spoofax language developers for specifying additional editor services or other transformations.

`index-lookup-one` looks for a definition of the given identifier in its owning scope. The `index-lookup` lookup performs a lookup that tries to look for a definition using `index-lookup-one`. If it cannot be found, the lookup is restarted on the outer scope until the root scope is reached. If no definition is found at the root scope, the lookup fails. There is also an `index-lookup-all` variant that returns all found definitions instead of stopping at the first found definition. Finally, `index-lookup-all-levels` is a special version of `index-lookup-all` that supports partial identifiers.

To get data from the index, `index-get-data` is used. Given a definition and a data kind, it will return all data values of that kind that is attached to the definition. Uses are retrieved in the same way using `index-get-uses-all`.

2.6.2 Reference resolution

Resolving a reference to its definition is very straightforward when using `index-lookup`, since it does all the work for us. The only thing that has to be done when Spoofax requests a reference lookup is a simple transformation: `node -> <index-lookup> node`. The resulting definition has location information embedded into it which is used to navigate to the reference. If the lookup fails, this is propagated back to Spoofax and no blue hyperlink will appear on the node under the cursor.

2.6.3 Constraint checking

Constraint checking rules are called by Spoofax after analysis on every AST node. If a constraint rule succeeds it will return the message and the node where the error marker should be put on.

The duplicate definition constraint check that was shown earlier is defined

```
constraint-error:
  node -> (key, "Duplicate definition")
  where
    <nam-unique> node ;
    key  := <nam-key> node ;
    defs := <index-lookup-one> key ;
    <gt> (<length> defs, 1)
```

Listing 2.13: Duplicate definitions constraint check written in Stratego, using generated code from the NaBL compiler and the index API.

```
editor-complete:
  ast -> identifiers
  where
    node@COMPLETION(name) := <collect-one(?COMPLETION(_))> ast ;
    proposals             := <index-lookup-all-levels(|name)> node ;
    identifiers           := <map(index-uri-name)> proposals
```

Listing 2.14: Code completion.

in listing 2.13. First `nam-unique` (generated for unique definitions by the NaBL compiler) is used to see if the node represents a unique definition; non-unique definition such as partial classes should not get duplicate definition error markers. The identifier is retrieved using `nam-key` and a lookup in the current scope is done with `index-lookup-one`. If more than one definition is found, the constraint check succeeds and an error marker is shown on the node.

2.6.4 Code completion

When code completion is requested in Spoofax, a completion node is substituted at the place where the cursor is. For example, if we request code completion on `VarRef("a")`, it will be substituted by `VarRef(COMPLETION("a"))` to indicate that the user wants to complete this identifier. See listing 2.14 for the code completion implementation. We first retrieve the completion node and name using `collect-one`. Completion proposals are gathered by `index-lookup-all-levels` since it can handle partial identifiers. Finally the retrieved proposals are converted to names by mapping `index-uri-name` over them.

2.7 Evaluation and Discussion

Our aim with this work has been to design high-level abstractions for name resolution applicable to a wide range of programming languages. In this section we discuss the limitations of our approach and evaluate its applicability to different languages and other language features than those covered in the preceding sections.

2.7.1 *Limitations*

There are two areas of possible limitations of NaBL. One is in the provided abstraction, the other is in the implementation algorithm that supports it. As for the provided abstraction, as a definition language, NaBL is inherently limited in the number of features it can support. While the feature space it supports is extensive, ultimately there may always be language features or variations that are not supported. For these cases, the definition of NaBL – written in Stratego – can be extended, or it is possible to escape NaBL and extend a specification using handwritten Stratego rules.

As for the implementation algorithm, NaBL's current implementation strategy relies on laziness, and does not provide much control over the traversal for the computation of names or types. In particular, sophisticated type inference schemes are not supported with the current algorithm. To implement such schemes, the algorithm would have to be extended, preferably in a way that maintains compatibility with the current NaBL definition language.

2.7.2 *Coverage*

During the design and construction of NaBL, we have performed a number of studies on languages and language features to determine the extent of the feature space that NaBL would support. In this paper we highlighted many of the features by using C# as a running example, but other languages that we studied include a subset of general-purpose programming languages C, Java, and domain-specific languages WebDSL [59], the Hibernate Query Language (HQL), and Mobl [61]. We also applied our approach to the Java Bytecode stack machine language using the Jasmin [104] syntax.

For our studies we used earlier prototypes of NaBL, which led to the design as it is now. Notable features that we studied and support in NaBL are partial classes, inheritance, visibility, lexical scoping, imports, type-based name resolution, and overloading; all of which have been discussed in section 2.3. In addition, we studied aspect-oriented programming with intertype declarations and pointcuts, file-based scopes in C, and other features.

Our design has also been influenced by past language definitions, such as SDF and Stratego. Altogether, it is fair to say that NaBL supports a wide range of language features and extensive variability, but can only support the full range of possible programming languages by allowing language engineers to escape the abstraction. In future work, we would like to enhance the possibilities of extending NaBL and design a better interface for escapes.

2.8 Related work

We give an overview of other approaches for specifying and implementing name resolution. The main distinguishing feature of our approach is the use of linguistic abstractions for name bindings, thus hiding the low level details of writing name analysis implementations.

2.8.1 Symbol Tables

In classic compiler construction, symbol tables are used to associate identifiers with information about their definition sites, typically including type information. Symbol tables are commonly implemented using hash tables where the identifiers are indexed for fast lookup. Scoping of identifiers can be implemented in a number of ways. For example, by using qualified identifiers as index, nesting symbol tables, or destructively updating the table during program analysis.

The type of symbol table influences the lookup strategy. When using qualified identifiers the entire identifier can be looked up efficiently, but considering outer scopes requires multiple lookups. Nesting symbol tables always requires multiple lookups but is more memory efficient. When destructively updating the symbol table, lookups for visible variables are very efficient, but the symbol table is not available after program analysis.

The index we use is a symbol table that uses qualified identifiers. We map qualified identifiers (URIs) to information such as definitions, types and uses.

2.8.2 Attribute Grammars

Attribute Grammars (AGs) [83] are a formal way of declaratively specifying and evaluating attributes for productions in formal grammars. Attribute values are associated with nodes and calculated in one or more tree traversals, where the order of computations is determined by dependencies between attributes.

Eli [74] provides an attribute grammar specification language for modular and reusable attribute computations. Abstract, language-independent computations can be reused in many languages by letting symbols from a concrete language inherit these computations. For example, computations `Range`, `IdDef`, and `IdUse` would calculate a scope, definitions, and references. A method definition can then inherit from `Range` and `IdDef`, because it defines a function and opens a scope. A method call inherits from `IdUse` because it references a function.

These abstract computations are reflected by naming concepts of NaBL and the underlying generic resolution algorithm. However, NaBL is less expressive, but more domain-specific. Where Eli can be used to specify general (and reusable) computations on trees, NaBL is restricted to name binding concepts, helping to understand and specify name bindings more easily.

Silver [163] is an extensible attribute grammar specification language which can be extended with general-purpose and domain-specific features. Typical examples are auto-copying, pattern matching, collection attributes, and support for data-flow analysis. However, name analysis is performed the traditional way: an environment with bindings is passed down the tree using inherited properties.

Reference Attribute Grammars (RAGs) extend AGs by introducing attributes that can reference nodes, substantially simplifying name resolution implementations.

JastAdd [53] is a meta-compilation system for generating language processors relying on RAGs and object orientation. It also supports parametrized

attributes to act as functions where the value depends on the given parameters. A typical name resolution as seen in [32, 53, 3] is implemented in lookup attributes parameterized by an identifier of use sites, such as variable references. All nodes that can have a variable reference as a child node, such as a method body, then have to provide an equation for performing the lookup. These equations implement scoping and ordering using Java code.

JastAdd implementations have much more low level details than NaBL declarations. This provides flexibility, but entails overhead on encoding and requires decoding for understanding. For example, scopes for certain program elements are encoded within a set of equations, usually implemented by early or late returns.

2.8.3 Visibility Predicates

CADET [112] is a notation for predicates and functions over abstract syntax tree nodes. Similar to attribute grammar formalisms, it allows to specify general computations in trees but lacks reusable concepts for name binding. Poetsch-Heffter proposes dedicated name binding predicates [119], which can be translated into efficient name resolution functions [118]. In contrast to NaBL, scopes are expressed in terms of start and end points and multi-file analyses are not supported.

2.8.4 Dynamic Rewrite Rules

In term rewriting, an environment passing style does not compose well with generic traversals. As an alternative, Stratego allows rewrite rules to create dynamic rewrite rules at run-time [17]. The generated rules can access variables available from their definition context. Rules generated within a rule scope are automatically retracted at the end of that scope.

Hemel et al. [60] describe idioms for applying dynamic rules and generic traversals for composing definitions of name analysis, type analysis, and transformations without explicitly staging them into different phases. Our current work builds on the same principles, but applies an external index and provides a specialized language for name binding declarations.

Name analysis with scoped dynamic rules is based on consistent renaming, where all names in a program are renamed such that they are unequal to all other names that do not correspond to the same definition site. Instead of changing the names directly in the tree, annotations can be added which ensure uniqueness. This way, the abstract syntax tree remains the same modulo annotations. Furthermore, unscoped dynamic rewrite rules can be used for persistent mappings [75].

2.8.5 Textual Language Workbenches

Xtext [12] is a framework for developing textual software languages. The Xtext Grammar Language is used to specify abstract and concrete syntax, but also name bindings by using cross-references in the grammar. Use sites are then automatically resolved by a simplistic resolution algorithm.

Scoping or visibility cannot be defined in the Grammar Language, but have

to be implemented in Java with help of a scoping API with some default resolvers. For example field access, method calls, and block scopes would all need custom Java implementations. Only package imports have special support and can be specified directly in the Grammar Language. Common constraint checks such as duplicate definitions, use before definition, and unused definitions also have to be specified manually. This increases the amount of boilerplate code that has to be rewritten for every language.

In contrast to Xtext's Grammar Language, NaBL definitions are separated from syntax definitions in Spoofax. This separation allows us to specify more advanced name binding concepts without cluttering the grammar with these concepts. It also preserves language modularity. When syntax definitions are reused in different contexts, different name bindings can be defined for these contexts, without changing the grammar. From an infrastructure perspective, Spoofax and Xtext work similarly, using a global index to store summaries of files and URIs to identify program elements.

EMFText [57] is another framework for developing textual software languages. Like Xtext, it is based on the Eclipse Modeling Framework [135] and relies on metamodels to capture the abstract syntax of a language. While in Xtext this metamodel is generated from a concrete syntax definition, EMFText takes the opposite approach and generates a default syntax definition based on the UML Human-Usable Textual Notation [65] from the metamodel. Language designers can then customize the syntax definition by adding their own grammar rules.

In the default setup, reference resolution needs to be implemented in Java. Only simple cases are supported by default implementations [58]. JastEMF [20] allows to specify the semantics of EMF metamodels using JastAdd RAGs by integrating generated code from JastAdd and EMF.

A Language Independent Task Engine for Incremental Name and Type Analysis

3

Abstract

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

3.1 Introduction

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

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

Incremental analyses are typically implemented manually. Thereby, change detection and dependency tracking are cross-cutting the implementation of the actual analysis. This raises complexity of the implementation and negatively affects maintenance, reusability, and modularity.

In this paper, we focus on incremental name and type analysis. We present a language-independent approach which consists of two phases. The first phase analyzes lexical scopes, collects information about binding instances, and creates deferred *analysis tasks* in a top-down traversal. An analysis task captures a single name resolution or type analysis step. Tasks might depend on other tasks and are evaluated in the second phase. Incrementality is supported on file level by the collection phase and on task level by the evaluation phase. When a file changes, only this file is recollected and only those tasks are reevaluated, which are affected by the changes in the collected data. As a consequence, the analysis does neither re-parse nor re-traverse unchanged files, even if they are affected by changes in other files. Only the affected analysis tasks are reevaluated.

Our approach enables language engineers to abstract over incrementality. When applied directly, language engineers need to parametrize the collection phase, where they have full freedom to create and combine low-level analysis tasks. Thereby, they can focus solely on the name binding and typing rules of their language while the generic evaluation phase provides the incrementality. The approach can also form the basis for more high-level meta-languages for specifying the static semantics of programming languages. We use the task engine to implement incremental name analysis for name binding and scope rules expressed in NaBL, Spoofax's declarative name binding language [89].

We have implemented the approach as part of the Spoofax language workbench [75] and evaluated it for WebDSL, a domain-specific language for the implementation of dynamic web applications [49], designed specifically to enable static analysis and cross-aspect consistency checking in mind [59]. We used real change-sets from the histories of two WebDSL applications to drive experiments for the evaluation of the correctness, performance and scalability of the obtained incremental static analysis. Experiment input data and the obtained results are publicly available.

We proceed as follows. In section 3.2 we introduce the basics of name and type analysis and introduce the running example of the paper. In sections 3.3 and 3.4, we discuss the two analysis phases of our approach, collection and evaluation. In section 3.5, we discuss the implementation and its integration into the Spoofax language workbench. In section 3.6, we discuss the evaluation of our approach. Finally, we discuss related work in section 3.7 and conclude in section 3.8.

3.2 Name and Type Analysis

In this section, we discuss name and type analysis in the context of the running example of the paper, a multi-file C# program shown in listing 3.1.

3.2.1 Name Analysis

In textual programming languages, an *identifier* is a name given to program elements such as variables, methods, classes, and packages. The same identifier can have multiple *instances* in different places in a program. Name analysis establishes relations between a *binding instance* that *defines* a name and a

```
class A {
  B b; int m;
  float m() {
    return 1 + b.f; }}
```

```
class B {
  int i; float f;
  int m() {
    return 0; }}
```

```
class C:A {

  int n() {
    return m(); }}
```

(a) Files before editing. The underlined expression causes a type error.

```
class A {
  B b; int m;
  int m(B b) {
    return 1 + b.i; }}
```

```
class B {
  int i; float f;
  int m() {
    return 1; }}
```

```
namespace N {
  class C:B {
    int n() {
      return m(); }}}
```

(b) Files after editing. Changes w.r.t. listing 3.1a are highlighted.

Listing 3.1: C# class declarations in separate files with cross-file references.

bound instance that *uses* that name [92]. Name analysis is typically defined programmatically through a name resolution algorithm that connects *binding prospects* to binding instances. When a prospect is successfully connected, it becomes a bound instance. Otherwise, it is a *free instance*.

The C# class declarations in listing 3.1a contain several references, some of which cross file boundaries. The declared type of field `b` in class `A` refers to class `B` in a separate file. Also, the return expression of method `m` in class `A` accesses field `f` in class `B`. The parent of class `C` refers to class `A` in a separate file and the return expression of method `n` in class `C` is a call to method `m` in class `A`.

Languages typically distinguish several *namespaces*, i.e. different kinds of names, such that an occurrence of a name in one namespace is not related to an occurrence of that same name in another. In the example, class `A` contains a field and a homonym method `m`, but C# distinguishes field and method names.

Scopes restrict the visibility of binding instances. They can be nested and name analysis typically looks for binding instances from inner to outer scopes. In the example, `b` is resolved by first looking for a variable `b` in method `A.m`, before looking for a field `b` in class `A`. A *named scope* is the context for a binding instance, and scopes other binding instances. In the example, class `A` is a named scope. It is the context for a class name and a scope for method and field names.

An *alias* introduces a new binding instance for an already existing one. An *import* introduces binding instances from one scope into another one. In the example, class `C` imports fields and methods from its parent class `A`.

3.2.2 *Type Analysis*

In statically typed programming languages, a *type* classifies program elements such as expressions according to the kind of values they compute [117]. Listing 3.1a declares method `C.n` of type `int`, meaning that this method is expected to compute signed 32-bit integer values. Type analysis assigns types to program elements. Types are typically calculated compositionally, with the type of a program element depending only on the types of its sub-elements [117].

Type checking compares expected with actual types of program elements. A

type error occurs if actual and expected type are incompatible. Type errors reveal at compile-time certain kinds of program misbehavior at run-time. In the example, the return expression in method `C.n` causes a type error. The expression is of type `float`, since the called method `m` returns values of this type. But the declaration of `C.n` states that it evaluates to values of type `int`. This will cause run-time errors, when a floating point value is returned by `C.n`, while an integer value is expected. Type analysis reveals this error early at compile-time.

In some cases, type analysis depends on name analysis. In the example of the return expression in `C.n`, `m` needs to be resolved in order to calculate the type of the return expression. In other cases, name analysis depends on type analysis. For example, the type of `b` needs to be calculated in order to resolve `f` in the return expression of `A.m`. In general, name resolution cannot only depend on types, but on a variety of *properties* of binding and bound instances.

3.2.3 Incremental Analysis

When a program changes, it needs to be reanalyzed. Different kinds of changes influence name and type analysis.

First, adding a binding instance may introduce bindings for free instances, or rebind bound instances. Removing a binding instance influences all its bound instances, which are either rebound to other binding instances or become free instances. Changing a binding instance combines the effects of removing and adding.

Second, adding a binding prospect requires resolution, while removing it makes a binding obsolete. Changing a binding prospect requires re-binding, resulting either in a new binding or a free instance.

Third, addition, removal, or change of scopes or imports influence bound instances in the affected scopes, which might be rebound to different binding instances or become free instances. Similarly, they influence bound instances which are bound to binding instances in the affected scopes.

Finally, addition of a typed element requires type analysis, while removing it makes a type calculation obsolete. Changing a typed element requires reanalysis.

Furthermore, changes propagate along dependencies. When bound instances are rebound to different binding instances or become free instances, this influences bindings in the context of these bound instances, the type of these instances, the type of enclosing program elements, and bindings in the context of such types. Consider listing 3.1b for an example. It shows edited versions of the C# class declarations from listing 3.1a. We assume the following editing sequence:

1. The return type of method `A.m` is changed from `float` to `int`. This affects the type of the return expression of method `C.n` and solves the type error, but raises a new type error in the return expression of `A.m`.
2. The return expression of method `A.m` is changed to `b.i`. This requires resolution of `i` and affects the type of the expression, solving the type error.

3. Parameter `B b` is added to method `A.m`. This might affect the resolution and by this the type of `b` and `i` in the return expression, the type of the return expression, the resolution of `m` in method `C.n`, and the type of its return expression. Actually, only the resolution of `b` and `m` and the type of the return expression in `C.n` are affected. The latter resolution fails, causing a resolution error and leaving the return expression untyped.
4. The parent of class `C` is changed from `A` to `B`. This affects the resolution of `m` in method `C.n` and the type of its return expression. It fixes the resolution error and the return expression becomes typed again.
5. Class `C` is enclosed in a new namespace `N`. This might affect the resolution of parent class `B`, the resolution of `m` in `N.C.n`, and the type of the return expression in `N.C.n`. Actually, it does not affect any of those.
6. The return expression of method `m` in class `B` is changed. This might affect the type of this expression, but actually it does not.

We discuss incremental analysis in the next sections. We start with the collection phase in section 3.3, and continue with the evaluation phase in section 3.4.

3.3 Semantic Index

We collect name binding information for all units in a project into a *semantic index*, a central data structure that is persisted across invocations of the analysis and across editing sessions. For the purpose of this paper, we model this data structure as binary relations over keys and values. As keys, we use URIs, which identify bindings uniquely across a project. As values, we use either URIs or terms. We use $\mathcal{U}$ and $\mathcal{T}$ to denote the set of all URIs and terms, respectively.

3.3.1 URIs

We assign a URI to each binding instance, bound instance, and free instance. A bound instance shares the URI with its corresponding binding instance. A URI consists of a language name, a list of scope segments, the namespace of the instance, its name, and an optional unique qualifier. This qualifier helps to distinguish unique binding instances by numbering them consecutively. A segment for a named scope consists of the namespace, the name, and the qualifier of the scoping binding instance. Anonymous scopes are represented by a segment `anon(u)`, where `u` is a unique string to distinguish different scopes. For example, `C#://Class.A.1/Method.m.1` identifies method `m` in class `A` in the C# program in listing 3.1a. The qualifier `1` distinguishes the method. Possible homonym methods in the same class would get subsequent qualifiers.

3.3.2 Index Entries

The index stores binding instances ($B \subseteq \mathcal{U} \times \mathcal{U}$), aliases ($A \subseteq \mathcal{U} \times \mathcal{U}$), transitive and non-transitive imports for each namespace ns ($TI_{ns} \subseteq \mathcal{U} \times \mathcal{U}$ and $NI_{ns} \subseteq \mathcal{U} \times \mathcal{U}$), and types of binding instances ($P_{type} \subseteq \mathcal{U} \times \mathcal{T}$). For a binding instance with URI u, B contains an entry (u', u), where u' is retrieved from u by omitting

Relation	Key	Value
B	C#:/Class.A	C#:/Class.A.1
	C#:/Class.A.1/Field.b	C#:/Class.A.1/Field.b.1
	C#:/Class.A.1/Field.m	C#:/Class.A.1/Field.m.1
	C#:/Class.A.1/Method.m	C#:/Class.A.1/Method.m.1
	C#:/Class.B	C#:/Class.B.1
	C#:/Class.B.1/Field.i	C#:/Class.B.1/Field.i.1
	C#:/Class.B.1/Field.f	C#:/Class.B.1/Field.i.1
	C#:/Class.B.1/Method.m	C#:/Class.B.1/Method.m.1
	C#:/Class.C	C#:/Class.C.1
	C#:/Class.C.1/Method.n	C#:/Class.C.1/Method.n.1
NI_{Field}, TI_{Field}	C#:/Class.C.1	Task:/31
NI_{Method}, TI_{Method}	C#:/Class.C.1	Task:/31
P_{type}	C#:/Class.A.1/Field.b.1	Task:/6
	C#:/Class.A.1/Field.m.1	int
	C#:/Class.A.1/Method.m.1	([], float)
	C#:/Class.B.1/Field.i.1	int
	C#:/Class.B.1/Field.f.1	float
	C#:/Class.B.1/Method.m.1	([], int)
	C#:/Class.C.1/Method.n.1	([], int)

Table 3.1: Initial semantic index for the C# program in listing 3.1a.

the unique qualifier. u' is useful to resolve binding prospects, as we will show later. An alias consists of the new name, that is a binding instance, and the old name, that is a binding prospect. For each alias, A contains an entry (a, u), where a is the URI of the binding instance and u is the URI of the binding prospect. For a transitive wildcard import from a scope with URI u into a scope with URI u', TI_{ns} contains an entry (u', u). Similarly, $NI_{\texttt{ns}}$ contains entries for non-transitive imports. Finally, for a binding instance of URI u and of type t, P_{type} contains an entry (u, t). P can also store other properties of binding instances, but we focus on types for this paper.

Example. Table 3.1 shows the index for the running example. It contains entries in B for binding instances of classes `A`, `B`, and `C`, fields `A.b`, `A.m`, `B.i`, and `B.f`, and methods `A.m`, `B.m`, and `C.n`. Corresponding entries for P_{type} contain the types of all fields and methods in the program. Since the running example does not define any aliases, A does not contain any entries. It also contains corresponding entries for NI_{Field}, TI_{Field}, NI_{Method}, and TI_{Method}. These entries model inheritance by a combination of a non-transitive and a transitive import. `C` first inherits the fields and methods from `A` (non-transitive import). Second, `C` inherits the fields and methods which are inherited by `A` (transitive import).

3.3.3 Initial Collection

We collect index entries in a generic top-down traversal, which needs to be instantiated with language-specific name binding and scope rules. During the

traversal, a dictionary S is maintained to keep track of the current scope for each namespace. At each node, we perform the following actions:

1. If the node is the context of a binding instance of name n in namespace ns, we create a new unique qualifier q, construct URIs $u' = S(ns)/ns.n$ and $u = u'.q$, and add (u', u) to B. If the instance is of type t, we add (u, t) to P_{type}. If the node is a scope for a namespace ns', we update $S(ns)$ to u.
2. If the current node is an anonymous scope for a namespace ns, we extend $S(ns)$ with an additional anonymous segment.
3. If the current node defines an alias, transitive, or non-transitive wildcard import, we add corresponding pairs of URIs to A, $TI_{\texttt{ns}}$, or $NI_{\texttt{ns}}$.

Collection does not consider binding prospects which need to be resolved. Furthermore, entries in $TI_{\texttt{ns}}$, $NI_{\texttt{ns}}$, and P_{type} might still require project-wide name resolution and type analysis. Instead of performing this analysis during the collection, we defer the remaining analysis tasks to a second phase of analysis and store unique placeholder URIs in the index. For example, the type of field `A.b` contains a class name `B`, which needs to be resolved. The index in table 3.1 does not contain an actual type, but a reference to a deferred resolution task (`Task:/6` in this case). Also, the index entries for wildcard imports refers to a deferred task, since the name of the base class of class `C` needs to be resolved first. References to tasks are created by hash consing, and therefore stay the same when the task stays the same.

Partitions. The semantic index is a project-wide data structure, but collection can be split over separate partitions. A *partition* is typically a file, but can also be a smaller unit. The only constraint we impose on partitions is that they need to be in global scope. This ensures that index collection is independent of other partitions. Collection for a partition p will provide us with a partial index consisting of B_p, A_p, $TI_{p,ns}$, $NI_{p,ns}$, and $P_{p,type}$. The overall index can be formed by combining all partial indices of a project.

3.3.4 Incremental Collection

When a partition is edited, reanalysis is triggered. But only the partial index of the changed partition needs to be recollected, while partial indices of other partitions remain valid. Partial recollection will result in an updated relation B'_p. Given the original B_p, we define a change set $\Delta_B = (B'_p \setminus B_p) \cup (B_p \setminus B'_p)$ of entries added to or removed from B. In the same way, we can define Δ_A, and $\Delta_{P_{type}}$. For imports, the situation is slightly different, since we need to consider changes in transitive import chains. We keep a change set $\Delta_{I_{ns}}$ for a derived relation $I_{ns} = TI^*_{ns} \circ NI_{ns}$, where TI^* is the reflexive transitive closure of TI and I is the composition of this closure with NI.

Example. Table 3.2 shows non-empty change sets for the running example, where superscripts indicate editing steps. In step 1, changing the return type of method `A.m` causes a change in P_{type}. In step 3, adding a parameter to the same method causes changes to B and P_{type}. In step 4, changing the parent of class `C` causes changes in I_{Field} and I_{Method}. In step 5, enclosing class `C` in a

Change	Key	Value
$\Delta^1_{P_{type}}$	C#:/Class.A.1/Method.m.1	~~([], float)~~
	C#:/Class.A.1/Method.m.1	([], int)
Δ^3_B	C#:/Class.A.1/Method.m.1/Var.b	C#:/Class.A.1/Method.m.1/Var.b.1
$\Delta^3_{P_{type}}$	C#:/Class.A.1/Method.m.1/Var.b.1	Task:/6
	C#:/Class.A.1/Method.m.1	~~([], int)~~
	C#:/Class.A.1/Method.m.1	([Task:/6], int)
$\Delta^4_{I_{Field}}$	C#:/Class.C.1	~~Task:/31~~
	C#:/Class.C.1	Task:/6
$\Delta^4_{I_{Method}}$	C#:/Class.C.1	~~Task:/31~~
	C#:/Class.C.1	Task:/6
Δ^5_B	C#:/Ns.N	C#:/Ns.N.1
	~~C#:/Class.C~~	~~C#:/Class.C.1~~
	C#:/Ns.N.1/Class.C	C#:/Ns.N.1/Class.C.1
	~~C#:/Class.C.1/Method.n~~	~~C#:/Class.C.1/Method.n.1~~
	C#:/Ns.N.1/Class.C.1/Method.n	C#:/Ns.N.1/Class.C.1/Method.n.1
$\Delta^5_{I_{Field}}$	~~C#:/Class.C.1~~	~~Task:/6~~
	C#:/Ns.N.1/Class.C.1	Task:/54
$\Delta^5_{I_{Method}}$	~~C#:/Class.C.1~~	~~Task:/6~~
	C#:/Ns.N.1/Class.C.1	Task:/54
$\Delta^5_{P_{type}}$	~~C#:/Class.C.1/Method.n.1~~	~~([], int)~~
	C#:/Ns.N.1/Class.C.1/Method.n.1	([], int)

Table 3.2: Changes to the semantic index of table 3.1, based on the changes to the C# program from listing 3.1b. Highlighted parts are modified or new values/entries. Striked through parts are old values or removed entries.

namespace affects all index entries for the class and its contained elements. The next section discusses how change-sets trigger reevaluation of deferred analysis tasks.

3.4 Deferred Analysis Tasks

In the previous section, we discussed the collection of index entries. This collection is efficient, since it requires only a single top-down traversal. When a partition changes, recollection is even more efficient, since it can be restricted to the changed partition, while the collected entries from other partitions remain valid. This is achieved by deferring name resolution and type analysis tasks, which might require information from other partitions or from other tasks.

Tasks are collected together with index entries and evaluated afterwards in a second analysis phase. For evaluation, no traversal is needed. Instead, inter-task dependencies determine an evaluation order. When a partition changes, only the tasks for this partition are recollected in the first phase. Change sets determine which tasks need to re-evaluated, including affected tasks from

Instruction	Semantics
`resolve uri`	$B\,[\texttt{uri}]$
`resolve alias uri`	$A\,[\texttt{uri}]$
`resolve import ns into uri`	$I_{\texttt{ns}}\,[\texttt{uri}]$
`lookup type of uri`	$P_{type}\,[\texttt{uri}]$
`check type t in T`	$\{\texttt{t}\} \cap \texttt{T}$
`cast type t to T`	$C\,[\texttt{t}] \cap \texttt{T}$
`assign type t`	$\{\texttt{t}\}$
`s1 + s2`	$R\,[\texttt{s1},\texttt{s2}]$
`s1 <+ s2`	$\begin{cases} R\,[\texttt{s1}] & \text{, if } \neq \emptyset \\ R\,[\texttt{s2}] & \text{, otherwise} \end{cases}$
`filter s1 + s2 by type T`	$\{u \in R\,[\texttt{s1},\texttt{s2}] \mid P_{type} \circ C\,[u] \cap \texttt{T} \neq \emptyset\}$
`filter s1 <+ s2 by type T`	$\begin{cases} \{u \in R\,[\texttt{s1}] \mid (P_{type} \circ C)\,[u] \cap \texttt{T} \neq \emptyset\} & \text{, if } \neq \emptyset \\ \{u \in R\,[\texttt{s2}] \mid P_{type} \circ C\,[u] \cap \texttt{T} \neq \emptyset\} & \text{, otherwise} \end{cases}$
`disambiguate s1 + s2 by type T`	$\{u \in R\,[\texttt{s1},\texttt{s2}] \mid \forall u' \in R\,[\texttt{s1},\texttt{s2}] : \delta_C(u',\texttt{T}) \geq \delta_C(u,\texttt{T})\}$
`disambiguate s1 <+ s2 by type T`	$\begin{cases} \{u \in R\,[\texttt{s1}] \mid \forall u' \in R\,[\texttt{s1},\texttt{s2}] : \delta_C(u',\texttt{T}) \geq \delta_C(u,\texttt{T})\} & \text{, if} \neq \emptyset \\ \{u \in R\,[\texttt{s2}] \mid \forall u' \in R\,[\texttt{s1},\texttt{s2}] : \delta_C(u',\texttt{T}) \geq \delta_C(u,\texttt{T})\} & \text{, ow.} \end{cases}$

Table 3.3: Syntax and semantics of name and type analysis instructions. `uri` denotes a URI, `ns` a namespace, `t` a type, `T` a set of types, and `s1`, `s2` subtask IDs.

other partitions.

3.4.1 Instructions

Each *task* consists of a special URI, which is used as a placeholder in the semantic index, its dependencies to other tasks, and an instruction. Table 3.3 lists the instructions which can be used in tasks. Their semantics is given with respect to the semantic index, a type cast relation $C \subseteq \mathcal{T} \times \mathcal{T}$, where $(t, t') \in C$ iff type t can be cast to type t', and a partial function $\delta_C : \mathcal{T} \times \mathcal{T} \to \mathbb{N}$ for the distance between types. We write $R\,[S]$ to denote the image of a set S under a relation R and omit set braces for finite sets, that is, we write $R\,[e]$ instead of $R\,[\{e\}]$. We provide three name resolution instructions for looking up binding instances from B (`resolve`), named imports from A (`resolve alias`), and wildcard imports from the derived relation I_{ns} (`resolve import`), and four type analysis instructions for type look-up from P_{type} (`lookup`), for checks with respect to expected types (`check`), for casts to an expected type according to C (`cast`), and for assigning types to program elements (`assign`).

ID	Instruction	Results
1	resolve C#:/Class.B	C#:/Class.B.1
2	resolve alias C#:/Class.B	
3	resolve Task:/2	
4	resolve import Class into C#:/	
5	resolve Task:/4/Class.B	
6	Task:/1 + Task:/3 + Task:/5	C#:/Class.B.1
7	assign type int	int
8	resolve C#:/Class.A.1/Method.m.1/Var.b	
9	resolve C#:/Class.A.1/Field.b	C#:/Class.A.1/Field.b.1
10	resolve import Field into C#:/Class.A.1	
11	resolve Task:/10/Field.b	
12	Task:/9 <+ Task:/11	C#:/Class.A.1/Field.b.1
13	Task:/8 <+ Task:/12	C#:/Class.A.1/Field.b.1
14	lookup type of Task:/13	C#:/Class.B.1
15	resolve Task:/14/Field.f	C#:/Class.B.1/Field.f.1
16	resolve import Field into Task:/14	
17	resolve Task:/16/Field.f	
18	Task:/15 <+ Task:/17	C#:/Class.B.1/Field.f.1
19	lookup type of Task:/18	float
20	check type Task:/7 in {int, long, float, double, String}	int
21	check type Task:/19 in {int, long, float, double, String}	float
22	cast type Task:/21 to Task:/20	
23	cast type Task:/20 to Task:/21	float
24	Task:/22 + Task:/23	float
25	cast type Task:/24 to float	float
26	cast type Task:/20 to int	int
27	resolve C#:/Class.A	C#:/Class.A.1
28	resolve alias C#:/Class.A	
29	resolve Task:/28	
30	resolve Task:/4/Class.A	
31	Task:/27 + Task:/29 + Task:/30	C#:/Class.A.1
32	resolve C#:/Class.C.1/Method.m	
33	resolve import Method into C#:/Class.C.1	C#:/Class.A.1
34	resolve Task:/33/Method.m	C#:/Class.A.1/Method.m.1
35	assign type []	[]
36	disambiguate Task:/32 <+ Task:/34 by type Task:/35	C#:/Class.A.1/Method.m.1
37	lookup type of Task:/36	([], float)
38	cast type Task:/37 to int	

Table 3.4: Tasks and their solutions for the C# program in listing 3.1a.

Example. Table 3.4 shows tasks and their solutions for the running example. Tasks 1 to 6 try to resolve class name `B`. Task 1 looks for `B` directly in the global scope. It finds an entry in *B* and *succeeds*. Task 2 looks for aliases, which task 3 tries to resolve next. Instead of a concrete URI, the task 3 has a reference to task 2. Since task 2 *fails* to find any named imports, task 3 also fails. Task 5 tries to resolve `B` inside imported scopes, which are yielded by task 4. Both tasks fail. Task 6 combines resolution results based on local classes, aliases, and imported classes. We will discuss such combinators in the next example.

Tasks 7 to 25 are involved in type checking the return expression of `A.m()` in listing 3.1a. Task 7 assigns type `int` to the integer constant. Tasks 8 to 18 are an example for the interaction between name and type analysis. The first six tasks try to resolve `b` either as a local variable, a field in the current class, or an inherited field. Next, task 14 looks up the type of the resolved field `A.b`, before the remaining tasks resolve field `f` with respect to that type `B`. Task 19 looks up the type of the referred field. The remaining tasks analyze the binary expression: Tasks 20 and 21 check if the subexpressions are numeric or string types. Tasks 22 and 23 try to coerce the left to the right type and vice versa. Both tasks are combined by task 24. Finally, task 25 checks if the type of the return expression can be coerced to the declared return type of the method.

3.4.2 Combinators

Table 3.3 also shows six instructions to combine the results of subtasks. The semantics of these combinators are expressed in terms of a relation R, where $(t, r) \in R$ iff r is a result of task t. Notably, tasks can have multiple results. We will revisit R later, when we discuss task evaluation.

The simplest combinators are a non-deterministic choice `+` and a deterministic pendant `<+`. The result of the non-deterministic choice is the union of the results of its subtasks. while the result of the deterministic choice is the result of its first non-failing subtask. Furthermore, we provide combinators `filter` and `disambiguate`. Both can be used in a non-deterministic or deterministic fashion to combine the result sets of resolution tasks with respect to expected types. `filter` keeps only compliant results. `disambiguate` keeps only results which fit best with respect to the expected types. The non-deterministic variant keeps all of them, while the deterministic variant chooses the first subtask which contributes to the best fitting results.

Example. In table 3.4, task 6 combines resolution results based on local classes, aliased classes, and imported classes. The non-deterministic choice ensures that no result is preferred over another. Similarly, task 24 combines the results of alternative coercion tasks. In tasks 12 and 13, deterministic choices ensure that local fields win over inherited fields and variables win over fields, respectively.

Method call resolution in the presence of overloaded methods is a well-known example for interaction between name and type analysis. Actual and formal argument types need to be considered by the resolution, since they need to comply. Furthermore, relations between these types indicate which declaration is more applicable. As an example, consider tasks 32 to 36 in table 3.4. They resolve method call `m()` in the return expression of `C.n()` from

listing 3.1a. Task 32 tries to resolve it locally, while tasks 33 and 34 consider inherited methods. Task 35 assigns an empty list as the type of the actual parameters of the call. Task 36 selects only these methods which fits this type best, preferring local over inherited methods. Finally, the last two tasks check the return expression of `C.n`. Task 37 looks up the type of `A.m`. Task 38 tries to casts this to the declared return type, but fails.

3.4.3 Initial Evaluation

During the generic traversal in the collection phase, we do not only collect semantic index entries but also instructions of tasks ($T \subseteq \mathcal{U} \times \mathcal{I}$) and inter-task dependencies ($D \subseteq \mathcal{U} \times \mathcal{U}$). Language-specific collection rules are needed to control the collection of name resolution and type analysis tasks. D imposes an evaluation order for tasks. First, we can evaluate independent tasks. Next, we can evaluate tasks which only depend on already evaluated tasks. This will evaluate all tasks except those with cyclic dependencies, which we consider erroneous. As mentioned earlier, we capture task results in a relation $R \subseteq \mathcal{U} \times (\text{URIs} \cup \mathcal{T})$.

Multiple Results. The instruction of each task is evaluated according to the semantics given in table 3.3. However, this only works, if we replace placeholders of dependent subtasks with their results. When a subtask has multiple results, we evaluate the dependent task for each of these results. Consider task 14 from table 3.4 as an example. It can only be evaluated after replacing the placeholder `Task:/13` with a result of the corresponding task. Since this task has a single result `C#:/Class.A.1/Field.b.1`, we actually need to evaluate the instruction `lookup type C#:/Class.A.1/Field.b.1`, yielding `C#://Class.B.1` as its only result.

3.4.4 Incremental Evaluation

When a partition is edited, the partial index and tasks for this partition will be recollected, resulting in an updated relation T'_p. We need to evaluate new tasks, which did not exist in another partition before. We collect the URIs of these tasks in a change set: $\Delta_{T_p} = \text{dom}(T'_p \setminus T_p)$. Furthermore, a changed semantic index might affect the results of the tasks from all partitions, requiring the reevaluation of those tasks. The various change sets determine which tasks need to be reevaluated:

$(u', u) \in \Delta_B$**:** tasks which evaluated an instruction `resolve` u'.

$(a, u) \in \Delta_A$**:** tasks which evaluated an instruction `resolve alias` a.

$(u', u) \in \Delta_I$**:** tasks which evaluated an instruction `resolve import` u'.

$(u, t) \in \Delta_{P_{type}}$**:** tasks which evaluated an instruction `lookup type of` u and `filter` or `disambiguate` tasks with a subtask s with $u \in R[s]$.

We maintain the URIs of these tasks in another change set Δ_T. The URIs of tasks which require evaluation is given by the set $\Delta_{T_p} \cup D^*[\Delta_T]$.

ID	Instruction	Results
39	`cast type Task:/24 to int`	
40	`resolve Task:/14/Field.i`	`C#:/Class.B.1/Field.i.1`
41	`resolve Task:/16/Field.i`	
42	`Task:/40 <+ Task:/41`	`C#:/Class.B.1/Field.i.1`
43	`lookup type of Task:/42`	`int`
44	`check type Task:/43 in {int, long, float, double, String}`	`int`
45	`cast type Task:/44 to Task:/20`	`int`
46	`cast type Task:/20 to Task:/44`	`int`
47	`Task:/45 + Task:/46`	`int`
48	`cast type Task:/47 to int`	`int`
49	`resolve C#:/Ns.N.1/Class.B`	
50	`resolve alias C#:/Ns.N.1/Class.B`	
51	`resolve Task:/50`	
52	`resolve import Class into C#:/Ns.N.1`	
53	`resolve Task:/52/Class.B`	
54	`Task:/49 + Task:/51 + Task:/53`	
55	`Task:/31 + Task:/54`	`C#/Class.B.1`
56	`resolve C#:/Ns.N.1/Class.C.1/Method.m`	
57	`resolve import Method into C#:/Ns.N.1/Class.C.1`	`C#:/Class.B.1`
58	`resolve Task:/57/Method.m`	`C#:/Class.B.1/Method.m.1`
59	`disambiguate Task:/56 + Task:/58 by type Task:/35`	`C#:/Class.B.1/Method.m.1`
60	`lookup type of Task:/59`	`([], int)`
61	`cast type Task:/60 to int`	`int`

Table 3.5: New tasks and their solutions for the C# program in listing 3.1b.

Example. In step 1 of the running example, task 25 becomes obsolete, since the return expression needs to be checked with respect to a new type, which is done by a new task 39, shown in table 3.5. Furthermore, the disambiguation in task 36 depends on an element in $\Delta^1_{P_{type}}$, which is to be reevaluated. Transitive dependencies trigger also the reevaluation of tasks 37 and 38. Since task 38 succeeds now, it does no longer indicate a type error in `C.n`. But the new task 39 fails, indicating a new type error in `A.m`.

In step 2, tasks 15, 17 to 19, 21 to 24, and 39 become obsolete, since another field needs to be resolved. The semantic index was not changed, and only the corresponding new tasks 40 to 48 need to be evaluated. In step 3, the additional variable parameter causes changes in the semantic index. Δ^3_B requires the reevaluation of task 8 and its dependent tasks 14, 16, and 40 to 48. Furthermore, $\Delta^3_{P_{type}}$ requires the reevaluation of task 36 and its dependent tasks 37 and 38. Similarly, $\Delta^4_{I_{Field}}$ requires the reevaluation of task 33 and its dependent tasks 34 and 36 to 38.

Finally, the new enclosing namespace introduced in step 5 makes tasks 32 to 34 and 36 to 38 obsolete and introduces new tasks 49 to 61, which take the new namespace into account.

3.5 Implementation

We have implemented the approach as four components of the Spoofax language workbench [75]. The first component is a Java implementation of the semantic index. It maintains a multimap storing relations B, A, I, and P, a set keeping partition names, and another multimap from partitions to their index entries. During collection, it calculates change sets on the fly, maintaining two multisets for newly added and removed elements.

The second component is a task engine implemented in Java. It maintains a map from task IDs to their instructions and bidirectional multimaps between task IDs and their partitions, between task IDs and index entries they depend on, and for task dependencies. Just as the semantic index, the task engine exposes a collection API and calculates change sets on the fly, maintaining a set of added and a set of removed tasks. Additionally, it exposes an API for task evaluation. During evaluation, it maintains a queue of scheduled tasks and a bidirectional multimap of task dependencies which are discovered dynamically. Results and messages of tasks are kept in maps. Both components use hash-based data structures which can be persisted to file. They support Java representations of terms as values and expose their APIs to Stratego [18], Spoofax's term rewriting language for analysis, transformation, and code generation.

The third component implements index and task collection as a generic traversal in Stratego. At each tree node, the traversal applies language-specific rewrite rules for name and type analysis. These rules can either be manually written in Stratego, or generated from meta-languages such as NaBL.

The fourth component is a compiler for NaBL, which generates language-specific rules for the index and task collection traversal. Additionally, we have created a new meta-DSL for specifying simple type systems, called TS, which also compiles to the index and task collection traversal. Therefore, language developers can specify their name and type analysis in NaBL and TS, from which a collection traversal is derived, which is then used to do incremental name and type analysis with the task engine. For example, listing 3.2 shows an extract of NaBL and TS rules for a C#-like language. If a certain name or type rule cannot be expressed in NaBL or TS, a collection rule can still be manually implemented by means of the API.

3.6 Evaluation

We evaluate the *correctness*, *performance*, and *scalability* of our approach with an implementation for name and type analysis of WebDSL programs. Correctness is interesting since we only analyze affected program elements. We expect incremental analysis to yield the same result as a full analysis. Performance and scalability are crucial since they are the main purpose of incremental

```
binding rules
  Class(NonPartial(), c, _, _): defines Class c scopes Field, Method
  Field(_, f)                  : defines Field f
  Method(_, m, _, _)           : defines Method m scopes Var

  Base(c):
    imports Field, imported Field, Method, imported Method from Class c

  ClassType(c)   : refers to Class c
  FieldAcc(e, f) : refers to Field f in Class c where e has type c
  VarRef(x)      : refers to Var x otherwise refers to Field x
  ThisCall(m, p*): refers to best Method m of type t* where p* has type t*
```

(a) Declarative name binding and scope rules in NaBL.

```
type rules
  Add(x, y) : ty
  where x : x-ty
    and y : y-ty
    and (
      (
            x-ty <is: String() and x-ty => ty
         or y-ty <is: String() and y-ty => ty
      )
      or
      (
            x-ty <is: Numerical() else error "Expected numerical" on x
        and y-ty <is: Numerical() else error "Expected numerical" on y
        and <promote-bin> (x-ty, y-ty) => ty
      )
    )
```

(b) Declarative type rule for string concatenation or numerical addition in TS.

Listing 3.2: Declarative name and type rules for a C#-like language.

analysis. We want to assess whether performance is acceptable for practical use in IDEs and how the approach scales for large projects. Specifically, we evaluate the following research questions: *RQ1*) Does incremental name and type analysis of WebDSL applications yield the same results as full analysis? *RQ2*) What is the performance gain of incremental name and type analysis of WebDSL applications compared to full analysis? *RQ3*) How does the size of a WebDSL application influence the performance of incremental name and type analysis? *RQ4*) Is incremental name and type analysis suitable for a WebDSL IDE?

3.6.1 Research method

In a controlled setting, we quantitatively compare the results and performance of incremental and full analysis of different versions of WebDSL applications. We have reimplemented name and type analysis for WebDSL, using NaBL to specify name binding and scope rules and Stratego to specify type analysis. We apply the same algorithm to perform full and incremental analyses to the

source code histories of two WebDSL applications. We run a full analysis on all files in a revision, and and incremental analysis only on changed files with respect to the result of a full analysis of the previous revision.

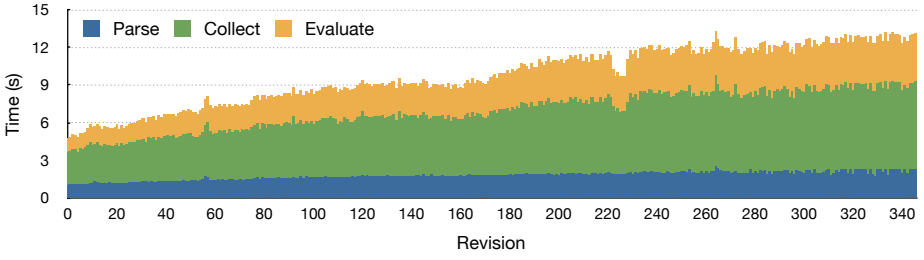
Subjects. WebDSL is a domain-specific language for the implementation of dynamic web applications [49]. It was designed from the ground up with static analysis and cross-aspect consistency checking in mind [59]. This focus makes it is an ideal candidate to evaluate its static analysis. WebDSL provides many language constructs on which constraints have to be checked. It also embodies a complex expression language that is representative of expressions in general purpose languages such as Java and C#. It has been used for several applications in production, including the issue tracker YellowGrass [149], which is a subject of this evaluation, the digital library Researchr, and the online education platform WebLab. When developing such larger applications, the usability of the WebDSL IDE sometimes suffered from the lack of incremental analyses.

We focus on two open source WebDSL applications: Blog [154], a web application for wikis and blogs, and YellowGrass [148], a tag-based issue tracker. In their latest revisions, their code bases consist of approximately 7 and 9 KLOC.

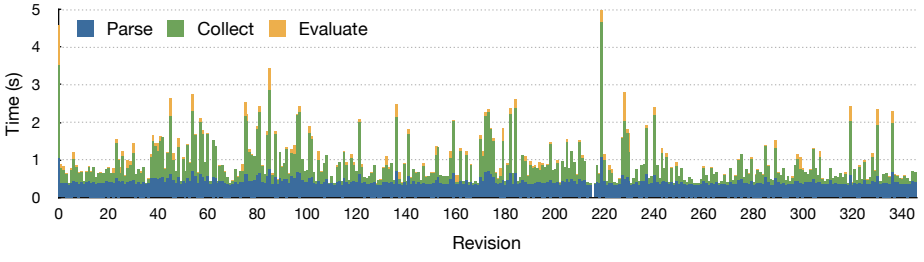
Data collection. We perform measurements by repeating the following for every revision of each application. We run an incremental and a full analysis. During each of the analyses we record execution timings. After each analysis we preserve the data from the semantic index and the task engine which we analyze afterwards.

Each analysis is sequentially executed on command line in a separate invocation of the Java Virtual Machine (JVM) and garbage collection is invoked before each analysis. After starting the virtual machine, we run three analyses and discard results allowing for the warmup period of the JVM's JIT compiler. All executions are carried out on the same machine with 2.7 Ghz Intel Core i-7, 16 GB of memory, and Oracle Java Hotspot VM version 1.6.0 45 in server JIT mode. We fix the JVM's heap size at 4 GB to decrease the noise caused by garbage collection. We set the maximum stack size at 16 MB.

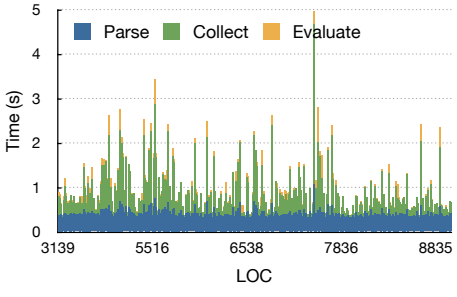
Analysis procedure. For *RQ1*, we evaluate the structural equality of data from the semantic index and the task engine produced by full and incremental analysis. For *RQ2*, we determine absolute execution times of full and incremental analysis and the relative speed up. We calculate the relative performance gain between analyses separately for each revision. We report geometric mean and distribution of absolute and relative performance of all revisions. For *RQ3*, we determine the number of lines and the number of changed lines of a revision. We relate the incremental analysis time to these numbers. For *RQ4*, we filter revisions which changed only a single file. On these revisions, we determine the execution time of incremental analysis.



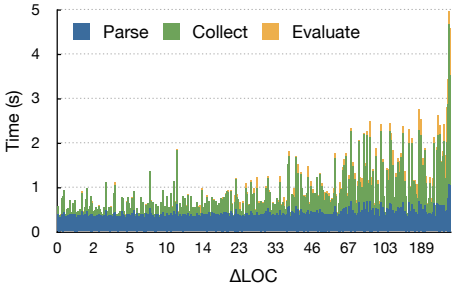
(a) Run time for full analysis, for each revision in the source code repository.



(b) Run time for incremental analysis, for each revision in the source code repository



(c) Incremental analysis time ordered by total Lines of Code (LOC).



(d) Incremental analysis time ordered by the size of the change: ΔLOC.

Figure 3.1: Benchmarking results

3.6.2 *Results and interpretation*

We published the collected data and all analysis results in a public repository [144], including instructions on reproducing our experiments. Since both applications yield similar results, we discuss only Yellowgrass data here. Data for Blog can be found in the repository. For the future, we plan to collect data on more WebDSL applications and on more programming languages. Our implementation and the subjects are also open source.

RQ1) For all revisions of both applications, incremental and full analysis produce structurally equal data in semantic index and task engine. This is the expected outcome and supports the equivalence of both analyses.

RQ2) Figures 3.1a and 3.1b show the absolute execution times of full and incremental analyses of all revisions. Full analysis takes between 4.74 and

13.31 seconds. Incremental analysis takes between 0.37 and 4.97 seconds. The mean analysis times are 9.75 seconds and 0.96 seconds, with standard deviations of 2.29 and 0.61 seconds, respectively. Incremental analysis takes between 3.06% and 43.75% of the time of a full analysis. The mean ratio between incremental and full analysis is 10.56%. Thus, incremental analysis gives huge performance gains.

RQ3) Figures 3.1c and 3.1d shows incremental analysis times per revision, ordered by LOC and changed LOC, respectively. The size of a project does not seem to influence incremental analysis time (correlation coefficient -0.18), but the size of the change does. This is the expected outcome, but more experiments will be needed

RQ4) There were 137 revisions which affected only a single file. Incremental analysis takes between 0.37 and 1.12 seconds. There is only one revision where incremental analysis takes longer than one second. The mean incremental analysis time is 0.56 seconds. All analysis times would be acceptable response times in an interactive IDE setting, where analysis is performed in the background without blocking the user interface. Single responses which take slightly more than one second would still be acceptable, if regular responses are fast. Furthermore, changes between two revisions are more coarse grained and should require more re-evaluation than changes in an editing scenario.

3.6.3 Threats to validity

An important threat to external validity is that we analyzed only WebDSL applications and only two of them. We are convinced that WebDSL's name and type analysis is representative for other languages, but our evaluation cannot generalize beyond WebDSL and its sublanguages. Furthermore, other WebDSL applications, particularly those of different size, might show different characteristics. Additional threats are the large distance between revisions and the correctness of revisions. In real-time editing scenarios, distances might be much smaller and revisions might switch between correct and erroneous states. We believe that smaller distances would only be in the benefit of incremental analysis. Erroneous revisions should not affect parse and collection times but evaluation times, which tend to be small. A threat to internal validity is file size. Incremental analysis re-parses and re-collects changed files. Independent of the actual changes inside a file, file size alone can influence parse and collection times. However, we believe that this does not influence the conclusions from any of our research questions. Regarding construct validity, we measured performance using wall-clock time only and control JIT compilation with a warm-up phase. By running the garbage collector between analysis runs, we ensured a similar amount of memory available to all analyses. However, the semantic index and the task engine store large amounts of data (13 MB in the worst case) and may experience garbage collection pauses.

3.7 Related Work

We give an overview of other approaches for incremental name and type analysis.

3.7.1 IDEs and Language Workbenches

IDEs such as Eclipse typically lack a generic framework for the development of incremental analyses, but provide manual implementations of incremental analysis and compilation for popular languages such as Java or C#. Some language workbenches automatically derive incremental analyses. In SugarJ [37], extensions inherit the incremental behaviour of SugarJ, which uses the module system of Java to provide incremental compilation on file-level, but lacks name and type analysis of its host language Java. Xtext [12] leverages incremental analysis and compilation from the Eclipse JDT to user-defined languages, as long as they map to Java concepts. The JDT performs only local analyses on edit and global analyses on save. MPS [159] does not require name binding due to its projectional nature. It supports incremental type analysis but lacks a framework for other incremental analyses. In general, language workbenches lack frameworks for developing incremental analyses.

3.7.2 Attribute Grammars

Attribute Grammars (AGs) [83] provide a formal way of specifying the semantics of a context-free language, including name and type analysis. One of the first incremental attribute evaluators is proposed in [25]. It only evaluates changed attributes and propagates evaluation to affected attributes. A similar incremental evaluation algorithm is shown in [164, 165] for ordered attributed grammars [73]. In [122, 124, 121, 72], extensions to propagation are shown that stop propagation if an attribute value is unchanged from its previous attribution.

Similar to attribute grammars, our approach exploits static dependencies, caching, and change propagation. Similar to ordered attribute grammars, we assume an evaluation order of tasks. Though tasks can be cyclic, we just do not evaluate them. While attributes are (re-)evaluated in visits to the tree, our collection separates tasks from the tree and they are (re-)evaluated independent of the tree. As a consequence, we do not require incremental parsing techniques and are not restricted to editing modes. For name analysis, attribute grammars typically pass environments throughout the tree. Incremental name analysis suffers from this as a single change in the environment requires a full re-evaluation of the aggregated environment and all dependent attributes. In our approach, we have a predefined notion of an environment, the semantic index, which is globally maintained. It enables fine-grained dependency tracking for name and type analysis tasks solely based on changing entries, not on changing environments.

3.7.3 Reference Attribute Grammars

A popular extension to attribute grammars is the addition of reference attributes. These simplify the specification of algorithms that require non-local information, including name resolution. Door Attribute Grammars [55, 54] extend attribute grammars with reference attributes and door objects which facilitate analysis of object-oriented languages. A similar but more general extension is shown in [120].

Reference Attribute Grammars (RAGs) [56] are a generalization of door attribute grammars where the door objects are removed. In [130], an incremental evaluator for reference attributed grammars is shown which is used by the JastAdd [33] meta-compilation system. JastAdd also adds parametrized attributes which allow attributes to be parametrized, forming a mapping. The approach is compared to traditional attribute grammars in [131] and shows that the use of reference attribute grammars reduces the number of affected attributes for name and type analysis significantly.

Our approach has two mechanisms similar to reference attributes. First, we can refer to binding instances by URIs and can look up their properties in the semantic index. Second, properties and tasks can refer to arbitrary other tasks. Reference attribute grammars discover dependencies during evaluation. We detect inter-task dependencies after collection. This already helps in establishing an ordering for evaluation. Only dependencies from properties to tasks are discovered during evaluation. Similar to ordinary attribute grammars, reference attribute grammars also do not provide a solution for aggregate attributes.

Some attribute grammar formalisms take a functional approach to evaluation. In [115] attributes are evaluated using visit-functions with memoization. A more general extension to attribute grammars is the higher order attribute grammar [157, 139] for which an incremental evaluator is presented in [156]. Similar to this approach, our approach employs a global cache and uses hash consing to efficiently share tasks and to make look-ups into the cache extremely fast. Tasks can also be seen as functions, but the evaluation strategy differs. Visit-functions are still applied on subtrees while tasks are completely separated from the tree.

3.7.4 Other Approaches

Pregmatic [13] is an incremental program environment generator that uses extended affix grammars for specification. It uses an incremental propagation algorithm similar to the one used by attribute grammar approaches which were discussed earlier. Instead of separating parsing and semantic analysis, all evaluation is done during parse-time which differs significantly from our parse, collect and evaluate approach. Incremental Rewriting [103] describes efficient algorithms for incrementally rewriting programs based on algebraic specifications. An algorithm for incrementally evaluating functions on aggregated values is also shown. The approach does not support non-local dependencies, making specification of name binding less intuitive as it requires copying of information.

3.8 Conclusion

We have proposed an approach for incremental name and type analysis in two phases, collection and deferred evaluation of analysis tasks. The collection is instantiated with language-specific name binding and type rules and incremental on file level. Unchanged files are neither re-parsed nor re-traversed. The evaluation phase is incremental on task level. When a file changes, all tasks

that are affected by this change are reevaluated. This might include dependent tasks from other files.

Tasks execute low-level instructions for name resolution and type analysis, and can form a basis for the definition of declarative meta-languages at a higher level of abstraction. For example, we map declarative name binding and scope rules expressed in NaBL to an instantiation of the presented approach. We implemented the approach as part of the Spoofax language workbench. It frees language engineers from the burden of manually implementing incremental analysis. We applied the implementation to WebDSL and empirical evaluation has shown this analysis to be responsive to changes in analyzed programs and suitable to the interactive requirements of an IDE setting.

Acknowledgements

This research was supported by NWO/EW Free Competition Project 612.001.114 (Deep Integration of Domain-Specific Languages) and by a research grant from Oracle Labs. We would like to thank Lennart Kats for his contribution to the start of NaBL and to Spoofax' incremental analysis project. We would also like to thank Karl Kalleberg for valuable discussions on the interpretation of name binding and scoping rules.

Reflection: Incremental Name and Type Analysis, Bootstrapping, and Spoofax Core

4

We reflect on the applications and expressiveness of NaBL, TS, and the task engine, which we will refer to as NaBL for brevity. Furthermore, we provide a vision for bootstrapping the meta-languages of Spoofax, and describe our work on Spoofax Core to make bootstrapping, other research, and application in industry feasible.

Incremental Name and Type Analysis

We have applied NaBL to develop the static analysis of Green-Marl [63], a DSL for graph analysis, initially developed at Stanford University, but now part of the Parallel Graph AnalytiX (PGX) [94] project at Oracle Labs. With Green-Marl, graph analysis programs can be concisely written in terms of graph analysis concepts such as nodes, vertices, properties of nodes and vertices, and depth/breadth-first traversals. The PGX runtime can then automatically parallelize these graph analyses.

We have also applied NaBL in the 2015-2016 edition of the Compiler Construction lab [145], where students develop a full version of the MiniJava [9] language, including name and type analysis and corresponding editor services.

Limitations in Expressiveness. NaBL can be used to model many interesting name and type analysis patterns, even for complicated DSLs such as Green-Marl. However, there are several limitations in expressiveness. For example, certain kind of let bindings such as `let*`, which are found in some functional languages, cannot be modeled because of a lack of control over scopes. Method overriding and overloading, where there are multiple method definitions a method call can refer to, and a selection must be made between those definitions based on parameter types, are not well supported. Finally, more comprehensive type systems with nonlocal inference are not supported.

Scope Graphs. Further work on improving the expressiveness has been done by other researchers in the context of Scope Graphs [109], a language-independent theory for program binding structure and name resolution. With this approach, a scope graph is first constructed from an abstract syntax tree using language-specific rules. Then, references in the scope graph are resolved to definitions using a language-independent resolution process. Name resolution is specified in terms of a concise, declarative and language-independent resolution calculus, while the actual name resolution is performed by a resolution algorithm that is sound and complete with respect to the calculus.

Hendrik van Antwerpen et al. refine and extend the scope graph framework to a full framework for static semantic analysis [5]. The framework is based

on a language of constraints, which support uniting type checking and name resolution. A language-specific extraction function translates an abstract syntax tree to a set of constraints, which are then solved by a constraint solver, which is proven to be sound. Although not discussed in [5], the extraction function can be generated from NaBL2, a meta-DSL for specifying name and type analysis in terms of syntax-directed constraint generation rules.

The expressiveness of the framework was further extended in [6] by viewing scopes as types, enabling models of the internal structure of non-simple types such as structural records and generic classes. Statix, a new meta-DSL was introduced, to enable specification of static semantics using this new framework.

The main difference between NaBL and scope graphs, is that scope graph approaches are more expressive, include the formal semantics of the calculi, and include algorithms that are sound in relation to the calculi. However, making these algorithms incremental and scalable; and providing good error messages, semantic completions, and other editor services; are still open problems.

Bootstrapping

We have self-applied NaBL and TS and the SDF [153] meta-DSL (for syntax specification), providing incremental name and type analysis and editor services for these meta-languages. These meta-languages all depend on each other. That is, NaBL's syntax is specified in SDF, its name binding in itself, and its type analysis in TS. Similarly, SDF and TS are specified in NaBL and each other. Therefore, we performed an ad-hoc form of *bootstrapping*, where for each meta-language, we apply its own compiler and the compiler of other meta-languages to the sources of the meta-language to generate code that implements parts of that meta-language. We then commit this generated code to source control, serving as a baseline for building the meta-language.

However, this ad-hoc form of bootstrapping is problematic, because committing generated (possibly binary) artifacts in source control creates additional load on source control storage. Furthermore, forgetting to commit the generated code may break the compilation of the meta-language. Finally, since we are applying all meta-language compilers only once, we cannot find cascading defects that are only evident after multiple applications.

Our vision to solve these bootstrapping problems is to do fixpoint bootstrapping, to version and separately store meta-language compilers (binaries), and to support using multiple versions of meta-language compilers simultaneously in an interactive system. We want to evaluate this approach by application to Spoofax and its meta-languages. However, using Spoofax as a vehicle for research turned out to be a problem for several reasons.

Spoofax Problem Analysis

First of all, Spoofax was dependent on the Eclipse IDE. It was not possible to run the Spoofax language workbench, or any language created with Spoofax,

```
private void registerWithImp(Language language) {
        final int TRIES = 10;
        final int SLEEP = 500;

        for (int i = 0; i < TRIES; i++) {
                try {
                        LanguageRegistry.registerLanguage(language);
                        return;
                } catch (ConcurrentModificationException e) {
                        // Loop
                        try {
                                Thread.sleep(SLEEP);
                        } catch (InterruptedException e2) {
                                throw new RuntimeException(e2);
                        }
                }
        }
}
```

Listing 4.1: Unclear concurrency and dynamic language loading in older versions of Spoofax.

outside of the Eclipse IDE, making it hard to develop, build, test, and deploy Spoofax.

One of the defining features of Spoofax is *dynamic language loading* [75], where a language under development can be dynamically reloaded in the IDE without restarting, with the changes being immediately visible. However, Spoofax used IMP [24] for its Eclipse editor services, which does not support dynamic language loading, and thus required a hack to get this working: a single IMP language which acted as all Spoofax dynamic languages, with the actual language being identified at runtime based on the extension of a file. Furthermore, there were two ways to load languages: statically from an Eclipse plugin, and dynamically at runtime, which resulted in different code paths for all editor services.

The concurrency model of Spoofax was unclear, leading to code as found in listing 4.1 in several places. Finally, Spoofax's code was very tightly coupled, making to modify a part without breaking other parts.

Besides these problems, we also missed two features crucial for bootstrapping meta-languages. We need to be able to version the compilers of languages, and to have explicit dependencies between languages. For example, to build NaBL version 2, we need SDF version 3's compiler, NaBL version 1's compiler, and TS version 1's compiler.

Spoofax Core

Given this problem analysis; and the need to do bootstrapping, other research, and better application in industry; we need to develop a better tooling foundation. Therefore, we have reimplemented Spoofax, dubbed *Spoofax Core*, which solves these problems and adds missing features. This was a joint effort

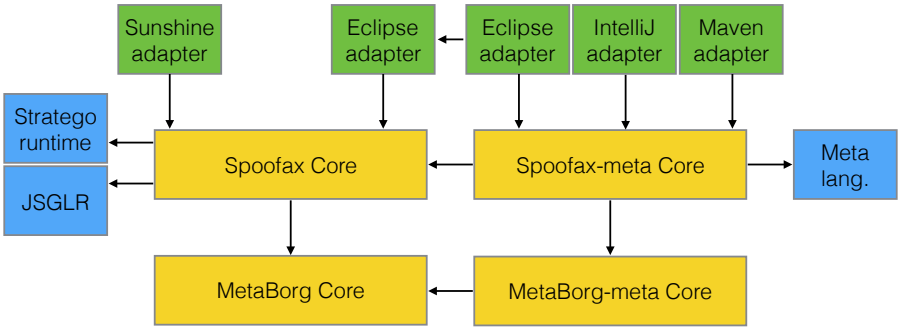


Figure 4.1: Spoofax Core component architecture.

with Vlad Vergu, Hendrik van Antwerpen, Daniel Pelsmaeker, Martijn Dwars, Eduardo Souza Amorim, and Sebastian Erdweg.

Spoofax Core is a platform-independent Java library, meaning that it can be integrated into any Java application, making custom integrations possible. We provide ready-made adapters for various platforms such as the command-line (called Sunshine), the Eclipse and IntelliJ IDEs, and the Maven build system. The language workbench environment, also called the meta-environment, is separated from the runtime environment so that languages can be used without a dependency on the meta-environment. Figure 4.1 is a depiction of this architecture

Spoofax Core can be developed, built, tested, and deployed from the command-line on Windows, Linux, and MacOS, allowing anyone to work on Spoofax. To ensure that Spoofax works, and to provide ready-made packages for using Spoofax, our Jenkins build farm continuously builds, tests, and deploys Spoofax Core and its adapters.

Spoofax Core is loosely coupled, enabling parts of Spoofax to be changed without breaking other parts. We use the Guice [48] dependency injection framework to hook up these loosely coupled parts as a separate concern, while also allowing several parts to be freely extended. Dynamic language loading is supported as a separate concern, with the rest of Spoofax only dealing with the concept of a language that is already loaded. Furthermore, we support concurrent environments such as IDEs by being thread-safe when its concurrency invariants are upheld. Finally, we have full support for versioning languages and explicit versioned dependencies between languages.

With this new foundation, we continue our research into bootstrapping in the next chapter (chapter 5).

Bootstrapping Domain-Specific Meta-Languages in Language Workbenches

5

Abstract

It is common practice to bootstrap compilers of programming languages. By using the compiled language to implement the compiler, compiler developers can code in their own high-level language and gain a large-scale test case. In this chapter, we investigate bootstrapping of compiler-compilers as they occur in language workbenches. Language workbenches support the development of compilers through the application of multiple collaborating domain-specific meta-languages for defining a language's syntax, analysis, code generation, and editor support. We analyze the bootstrapping problem of language workbenches in detail, propose a method for *sound* bootstrapping based on fixpoint compilation, and show how to conduct breaking meta-language changes in a bootstrapped language workbench. We have applied sound bootstrapping to the Spoofax language workbench and report on our experience.

5.1 Introduction

A bootstrapped compiler can compile its own source code, because the compiler is written in the compiled language itself. Such bootstrapping yields four main advantages:

1. A bootstrapped compiler can be written in the compiled high-level language.
2. It provides a large-scale test case for detecting defects in the compiler and the compiled language.
3. It shows that the language's coverage is sufficient to implement itself.
4. Compiler improvements such as better static analysis or the generation of faster code applies to all compiled programs, including the compiler itself.

Compiler bootstrapping is common practice nowadays. For example, the GCC compiler for the C language is a bootstrapped compiler; its source code is written in C and it can compile itself. More generally for a language $\mathtt{L}$, a bootstrapped compiler $\mathtt{L_c}$ should apply to its own definition $\mathtt{L_d}$ such that $\mathtt{L_d} \in \mathtt{L}$ and $\mathtt{L_c}(\mathtt{L_d}) = \mathtt{L_c}$.

Language workbenches [39] are compiler-compilers that provide high-level meta-languages for defining domain-specific languages (DSLs) and their compilers. Thus, users of a language workbench implement the compiler $\mathtt{L_c}$ of language $\mathtt{L}$ not in $\mathtt{L}$ but in a high-level meta-language $\mathtt{M}$ such that $\mathtt{L_d} \in \mathtt{M}$ and $\mathtt{M_c}(\mathtt{L_d}) = \mathtt{L_c}$. Thus, bootstrapping of $\mathtt{L_c}$ is no longer required, which is good since many DSLs have limited expressiveness and are often ill-suited for

compiler development.

What we desire instead is bootstrapping of a language workbench's compiler-compiler $\mathtt{M_c}$. We want to use our meta-languages for implementing our meta-language compilers, thus inheriting the benefits of bootstrapping stated above: high-level meta-language implementation, large-scale test case, meta-language coverage, and improvement dissemination. In short, bootstrapping of language workbenches supports meta-language development. However, bootstrapping of language workbenches also entails three novel challenges:

- Most language workbenches provide separate meta-languages $\mathtt{M}^{1..n}$ for describing the different language aspects such as syntax, analysis, code generation, and editor support. Thus, to build the defintion of any one meta-language compiler $\mathtt{M}^i_\mathtt{d}$, multiple meta-language compilers $\mathtt{M}^{1..n}_\mathtt{c}$ are necessary such that $\mathtt{M}^{1..n}_\mathtt{c}(\mathtt{M}^i_\mathtt{d}) = \mathtt{M}^i_\mathtt{c}$. This entails intricate dependencies that sound language-workbench bootstrapping needs to handle.
- Most language workbenches provide an integrated development environment (IDE). Typically, language workbenches generate or instantiate this IDE based on the definition of the meta-languages. In this setup, the meta-language developer needs to restart the IDE whenever the definition of a meta-language is changed. However, to support bootstrapping, the definition of meta-language compilers should be available *within* the IDE and no restart should be required to generate and load the new bootstrapped meta-language compilers [88]. Importantly, since meta-language changes can be defective, it also needs to be possible to rollback to an older meta-language version if bootstrapping fails.
- Since meta-languages in language workbenches depend on one another, it can become difficult to implement breaking changes that require the simultaneous modification of a meta-language and existing client code. For example, renaming a keyword in one meta-language can require modifications in the compilers of the other meta-languages. To preserve changeability, we need to support implementing such breaking changes in a bootstrapped language workbench.

We present a solution to these challenges based on versioning and fixpoint bootstrapping of meta-language compilers. That is, we iteratively self-apply meta-language compilers to derive new versions until no change occurs. For this to work, we identified properties that meta-language compilers need to satisfy: explicit cross-language dependencies, deterministic compilation, and comparability of compiler binaries. To support meta-language engineers, we describe how to build interactive environments on top of fixpoint bootstrapping. Finally, we discuss how to implement and bootstrap breaking changes in the context of fixpoint bootstrapping.

To confirm the validity of our approach, we have implemented fixpoint bootstrapping for the Spoofax language workbench [75]. We use our implementation to successfully bootstrap eight meta-languages. We present our experience with seven changes to the meta-languages. We describe how we implemented the changes, how bootstrapping helped us to detect defects, and

how we handled breaking changes.

We are the first to describe a method for bootstrapping the meta-languages of a language workbench. In summary, we make the following contributions:

- We present a detailed problem analysis and requirements for language-workbench bootstrapping (section 5.2).
- We describe a sound bootstrapping method based on fixpoint meta-language compilation (section 5.3).
- We explain how to build bootstrapping-aware interactive environments (section 5.4).
- We investigate support for implementing breaking changes in a bootstrapped language workbench (section 5.5).
- We validate our approach by realizing it in Spoofax and by investigating seven bootstrapping changes (section 5.6).

5.2 Problem Analysis

To get a better understanding of bootstrapping in the context of language workbenches, we analyze the problem of bootstrapping in more detail. This problem analysis will help us answer why we need bootstrapping in the first place, and what is required to do bootstrapping in the context of language workbenches.

5.2.1 Bootstrapping Example

First, we need a more realistic example that shows the complexities of bootstrapping language workbenches. As an example, we use the SDF and Stratego meta-languages from the Spoofax language workbench. SDF [153] is a meta-language for specifying syntax of a language. Stratego [18] is a meta-language for specifying term transformations. SDF and Stratego are bootstrapped by self specification and mutual specification. That is, SDF's syntax is specified in SDF, and its transformations in Stratego. Stratego's syntax is specified in SDF, and its transformations in Stratego.

SDF also contains several generators. SDF contains a pretty-printer generator `PP-gen` that generates a pretty-printer based on the layout and concrete syntax in a syntax specification [158]. A pretty-printer (sometimes called an unparser) is the inverse of a parser. It takes a parsed abstract syntax tree (AST) and pretty-prints it back to a string. The generated pretty-printer is a Stratego program that performs this function. Besides generating a pretty-printer, SDF contains a signature generator `Sig-gen` that generates signatures for the nodes occurring in the AST. Since these signatures serve as a basis for AST transformations in Stratego, SDF describes these signatures in Stratego syntax and pretty-prints them using the generated Stratego pretty-printer.

Overall, our scenario entails various complex dependencies across languages. In the remainder of this section, we focus on the following dependency chain:

- The pretty-printer generator translates SDF ASTs into Stratego ASTs and

thus requires the SDF and Stratego signatures.

- The SDF signatures are generated by the signature generator using the Stratego pretty-printer.
- The Stratego pretty-printer is generated by the pretty-printer generator, from the Stratego syntax definition.
- The pretty-printer generator is implemented as a Stratego program within SDF.

We want to apply bootstrapping to SDF and Stratego to detect defective changes to a language's implementation. In order to illustrate the difficulties of bootstrapping in the context of language workbenches, we will deliberately introduce a defect in the implementation of the pretty-printer generator. Normally, a pretty-printer needs to align with the parser such that $\mathit{parse}(\mathit{pretty\text{-}print}(\mathit{ast})) = \mathit{ast}$. We break the the pretty-printer generator to violate this equation by generating pretty-printers that print superfluous semicolons. This is an obvious way to sabotage the pretty-printer generator and will cause parse failures when parsing a pretty-printed string. We expect bootstrapping to detect this defect. Figure 5.1 shows an iterative bootstrapping attempt with relevant dependencies, illustrating code examples, and an explanation for each bootstrapping iteration.

We start with a baseline of language implementations. We introduce the defect in the pretty-printer generator and start rebuilding the whole system in Iteration 1 using the baseline. However, despite the defect, all components build fine in Iteration 1. This is because it takes multiple iterations for the defect to propagate through the system before it produces an error. In our example, the defective pretty-printer generator (Iteration 1) generates a broken pretty-printer (2), which is used by the signature generator (3), which then generates signatures in Stratego syntax but with superfluous semicolons (4). All defects remain undetected until the build of `PP-gen` or `Sig-gen` in Iteration 4 fails because of parse errors in the signatures.

Our example illustrates multiple points. First, dependencies between components in a language workbench are complex, circular, and across languages. Second, language bootstrapping yields a significant test case for language implementations and can successfully detect defects. Third, a single build is insufficient because many defects only materialize after multiple iterations of rebuilding.

This example is still far removed from the complexity that language workbenches face in practice. For example, Spoofax features eight interdependent meta-languages and SDF alone has seven generators that uses pretty-printers from four other meta-languages.

5.2.2 Requirements

Based on our example above, we derive requirements for sound bootstrapping support in language workbenches.

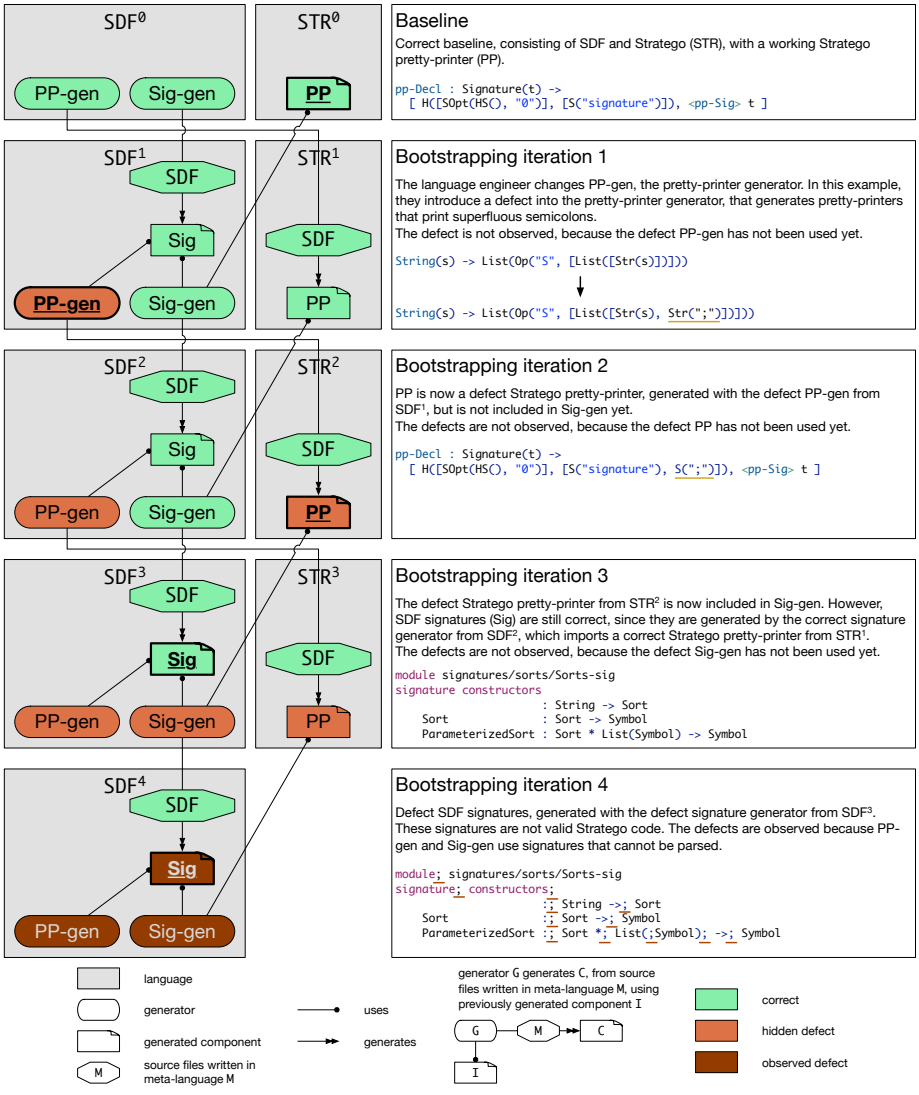


Figure 5.1: Bootstrapping flow for bootstrapping SDF and Stratego with a defective pretty-printer generator. In each iteration, SDF and Stratego are compiled, based on their previous versions. For example, $\texttt{SDF}^2$ is compiled with $\texttt{SDF}^1$ and $\texttt{STR}^1$. In the fourth iteration, bootstrapping fails because of a parse error, which can be traced back to the change which introduces a defect into the pretty-printer generator in the first iteration. Source code on the right belongs to underlined/bold components on the left.

Sound Bootstrapping. In our example, we needed 4 bootstrapping iterations to find a failure caused by the defective pretty-printer. In general, there is no way to know how many iterations are necessary until a defect materializes or after how many iterations it is safe to stop. Therefore, for sound bootstrapping it is required to iterate until reaching a fixpoint, that is, until the build stabilizes.

To determine if a fixpoint has been reached, we must be able to compare the binaries that meta-languages generate. We have reached a fixpoint if the generated binaries in iteration $k+1$ are identical to the binaries generated in iteration k. Since the binaries are the same, further rebuilds after reaching a fixpoint cannot change the implementation or detect new defects.

A further requirement for fixpoint bootstrapping is that compilers must be deterministic. That is, when calling a compiler with identical source files, the compiler must produce identical binaries.

Bootstrapping always requires a baseline of meta-language binaries to kick-start the process. Bootstrapping uses the baseline only to rebuild the meta-languages in the first bootstrapping iteration. After that, bootstrapping uses the bootstrapped binaries.

Finally, the bootstrapping system should be general; it should work for any meta-language in the language workbench.

Interactive Bootstrapping Environment. Besides having a bootstrapping system that satisfies the requirements above, we also need to support bootstrapping in the interactive environments of language workbenches. In particular, an interactive environment needs to provide operations that (1) start a bootstrapping attempt, (2) load a new baseline into the environment after bootstrapping succeeded, (3) roll back to an existing baseline after bootstrapping failed, and (4) cancel non-terminating bootstrapping attempts.

Loading a baseline needs to be such that subsequent bootstrapping attempts use the new baseline. When bootstrapping fails, a rollback to the existing baseline is required such that the defect causing the failure can be fixed and a new bootstrapping attempt can be started. All operations should work within the same language workbench environment, without requiring a restart of the environment, or a new environment to be started.

Bootstrapping Breaking Changes. Bootstrapping helps to detect changes that break a language implementation. However, sometimes breaking changes are desirable, for example, to change the syntax of a meta-language. If we change the syntax definition of some language L and the code written in L simultaneously, bootstrapping fails to parse the source code in Iteration 1 because the baseline only supports the old syntax of L. If we change the syntax definition of L but leave the code written in L unchanged, bootstrapping fails to parse the source code in Iteration 2 because the parser of L generated in Iteration 1 only supports the new syntax of L.

The bootstrapping environment should provide operations for bootstrapping breaking changes.

5.3 Sound Bootstrapping

Compiling or bootstrapping a meta-language is a complex operation that requires application of generators from many meta-languages, to a meta-language that consist of sources in several meta-languages. Therefore, we would like to find a general compilation and bootstrapping algorithm.

We describe a method for *sound bootstrapping* that fulfills the requirements from the previous section. As a first step towards compilation and bootstrapping, we introduce a general model for meta-language definitions and products. Using the model, we describe a general compilation algorithm that compiles a meta-language definition into a meta-language product. Finally, we show how to perform fixpoint bootstrapping operations based on the model and compilation algorithm. We use the bootstrapping scenario from fig. 5.1 as a running example in this section.

5.3.1 Language Definitions and Products

As a first step towards bootstrapping, we introduce a general model for language definitions and products. We require such a model to describe a general compilation and bootstrapping algorithm for meta-languages. Listing 5.1 shows the model encoded in Haskell. In this subsection, we explain the model. In later subsections, we explain the compilation and fixpoint bootstrapping algorithm.

Language. First of all, we use a *unique name* to identify each meta-language `Lang` of a language workbench, such as SDF and Stratego. However, a name alone is not enough to uniquely distinguish meta-languages. Multiple versions of the same meta-language exist when bootstrapping, for example, a baseline version of SDF and the first bootstrapping iteration of SDF. Therefore, we also use a *version* to identify a language, `LangID` in the model. We denote a language L with version 1 as $\mathtt{L}^1$. For example, with versioning, we can uniquely identify different versions of SDF and Stratego: $\mathtt{SDF}^0$, $\mathtt{STR}^0$, $\mathtt{SDF}^1$, $\mathtt{STR}^1$.

Bootstrapping applies the generators of a meta-language to the definition of its own and other meta-languages. Therefore, it is important to distinguish a meta-language *definition* from a meta-language *product*, which results from compiling the definition. The example in fig. 5.1 does not make this distinction to reduce its complexity, but we require this distinction here in order to precisely define compilation and bootstrapping.

Language Definition. Each language definition `LangDef` defines a specific version of a language. We denote the definition of language L at version 1 as $\mathtt{L}^1_\mathtt{d}$. The definition consists of source artifacts written in different meta-languages (field `alang` of `Artifact`). To compile a language definition, we need to know what external artifacts and generators it requires. To this end, a language definition defines artifact and generator dependencies on previous versions of itself or on specific versions of other languages. We use these dependencies during compilation.

```haskell
-- Model for languages, language definitions with sources, and language products with
   artifacts and generators.
type Version = Int
type Lang = String
data LangID = LangID { name :: Lang, version :: Version }
data Artifact = Artifact { aname :: String, alang :: Lang, acontent :: String }
data LangDef = LangDef { dlang :: LangID, dsources :: [Artifact], dartDeps :: [LangID],
  dgenDeps :: [LangID] }
data Generator = Generator { gname :: String, gsource :: Lang, gtarget :: Lang,
  generate :: Artifact -> Artifact }
data LangProd = LangProd { plang :: LangID, partifacts :: [Artifact],
  pgenerators :: [Generator] }
type Baseline = [LangProd]

getProd :: LangID -> Baseline -> LangProd
getProd lang baseline = fromJust $ find (\prod -> lang == plang prod) baseline

-- Compile. Sort languages by generator source/target and run relevant generators against
   relevant artifacts.
compile :: LangDef -> Baseline -> LangProd
compile def baseline = createLangProd def (runGens sorted gens inputs) where
  inputs = dsources def ++ [ a | l <- dartDeps def, a <- partifacts (getProd l baseline) ]
  gens = [ g | l <- dgenDeps def, g <- pgens (getProd l baseline) ]
  sorted = topsort [ l | LangID l _ <- dgenDeps def ] [ (gsource g,gtarget g) | g <- gens]

runGens :: [Lang] -> [Generator] -> [Artifact] -> [Artifact]
runGens [] gens inputs = inputs
runGens (lang:langs) gens inputs = runGens langs gens (inputs ++ runGensFor lang gens
   inputs)

runGensFor :: Lang -> [Generator] -> [Artifact] -> [Artifact]
runGensFor lang gens inputs = [ generate g a | g <- gens, a <- inputs, gsource g == lang,
   alang a == lang ]

createLangProd :: LangDef -> [Artifact] -> LangProd -- Implemented by the LWB.

-- Fixpoint bootstrap language definitions with a baseline.
-- Update versions in the first iteration, then fixpoint.
bootstrap :: Version -> [LangDef] -> Baseline -> (Baseline, [LangDef])
bootstrap v defs baseline =
  let firstBuild = [ compile (setVersion v def) baseline | def <- defs ] in
    bootstrapFixpoint (prepareFixpoint v defs) firstBuild

bootstrapFixpoint :: [LangDef] -> Baseline -> (Baseline, [LangDef])
bootstrapFixpoint defs baseline =
  let newBaseline = [ compile def baseline | def <- defs ] in
    if baseline == newBaseline
    then (newBaseline, defs)
    else bootstrapFixpoint defs newBaseline

setVersion :: Version -> LangDef -> LangDef
setVersion v (LangDef (LangID l _) srcs gdeps adeps) = LangDef (LangID l v) srcs gdeps
   adeps

prepareFixpoint :: Version -> [LangDef] -> [LangDef]
prepareFixpoint v defs = [ prepareFixpointDef v bootstrappedLangs def | def <- defs ]
  where bootstrappedLangs = [ l | LangDef (LangID l _) _ _ _ <- defs ]

prepareFixpointDef :: Version -> [Lang] -> LangDef -> LangDef
prepareFixpointDef v langs (LangDef (LangID l _) srcs adeps gdeps) =
  LangDef (LangID l v) srcs [ updateDep v langs dep | dep <- adeps ] [ updateDep v langs
     dep | dep <- gdeps ]

updateDep :: Version -> [Lang] -> LangID -> LangID
```

Listing 5.1: Model for sound bootstrapping, with algorithms for compilation and fixpoint bootstrapping, encoded in Haskell.

Language Product. A language product `LangProd` models a compiled meta-language definition. We denote the product of compiling $\texttt{L}^1_\texttt{d}$ as $\texttt{L}^1_\texttt{p}$. A product exports artifacts and generators. A generator `Generator` transforms artifacts of some source language into artifacts of some target language. For example, `Sig-gen` in SDF transforms SDF artifacts into Stratego artifacts, or fails if the SDF artifacts are invalid, which we model as a dynamic exception of function `generate`.

Example. Language definitions and products model the dependencies required to compile a definition into a product, which we describe in the next subsection. For example, $\texttt{SDF}^1_\texttt{d}$ requires the application of generator `Sig-gen` of $\texttt{SDF}^0_\texttt{p}$, whereas $\texttt{STR}^1_\texttt{d}$ requires the application of generator `PP-gen` of $\texttt{SDF}^0_\texttt{p}$. Moreover, $\texttt{SDF}^1_\texttt{d}$ requires the pretty-print table artifact `PP` of $\texttt{STR}^0_\texttt{p}$.

5.3.2 Compilation

Before we can bootstrap multiple meta-language definitions against a baseline of meta-language products, we must first be able to compile a single meta-language definition. We describe the compilation algorithm that compiles a single language definition using the model from above.

Function `compile` takes a language definition and a baseline of language products, and produces a new language product from the definition. The basic idea of the algorithm is to run the required generators on the source artifacts and the required external artifacts. This yields new generated artifacts that we package into a language product using `createLangProd`.

We first collect all generator inputs, which are the source artifacts `(dsources def)` of the definition and the required artifacts according to dependencies `(dartDeps def)`. We use the baseline to resolve dependencies; function `getProd` finds the product of the required `LangID`. Similar to required artifacts, we collect the required generators according to dependencies `(dgenDeps def)`.

When running generators, we need to make sure to call them in the right order: A generator must run later if it consumes an artifact produced by another generator. For example, $\texttt{SDF}^1_\texttt{d}$ requires the application of generator `Sig-gen`, which produces Stratego code. But $\texttt{SDF}^1_\texttt{d}$ also requires the application of the Stratego-compiler generator of $\texttt{STR}^0_\texttt{p}$, which translates Stratego code into an executable. Thus, we must run `Sig-gen` before the Stratego compiler. To this end, we sort all languages topologically according to their source and target languages.

Function `runGenerators` iterates over the sorted source languages and for each one applies all generators of the current source language `lang`. Function `runGeneratorsFor` finds all relevant generators `g` that take artifacts of `lang` as input and it finds all relevant artifacts `a` of `lang`. It then calls the `generate` function of all relevant generators `g` on all relevant artifacts `a` and collects and returns the generated artifacts. Function `runGenerators` passes the generated artifacts down when recursing to allow subsequent generators to compile them. If any `generate` function fails with a dynamic exception, the compilation fails.

Finally, after generating all artifacts, we create a language product from the

language definition and the generated artifacts by calling `createLangProd`. This function must be implemented by the language workbench. We abstract over how a language workbench determines which artifacts to export and which generators to create from generated artifacts. For example, Spoofax determines which artifacts to export from a configuration file in the language definition, has built-in notions of generators to create based on generated artifacts, and allows a language definition to configure its own generators.

5.3.3 *Fixpoint Bootstrapping*

We can now use compilation to define fixpoint bootstrapping. In general, there is no way to know how many bootstrapping iterations are required before it is safe to stop. Therefore, we iteratively bootstrap meta-languages until reaching a fixpoint. We define a general fixpoint bootstrapping algorithm using the model and compilation algorithm from above.

Function `bootstrap` takes the version of the new baseline, a list of meta-language definitions, and an existing baseline, and it produces a new baseline of the given version. The basic idea of the algorithm is to compile meta-language definitions in iterations, until we reach a fixpoint. However, to avoid building against the old baseline repeatedly, we have to update the versions of the language definitions in the first iteration.

In the first iteration, function `bootstrap` calls `compile` on modified definitions `def` where we have set the version to `v`. This yields a list of language products `firstBuild` that contains products of version `v`. We use this as starting point for fixpoint computation. In addition, we update the dependencies in `defs` using function `prepareFixpoint`, which updates versions and dependencies of all bootstrapped languages to `v`.

To produce a new baseline, we repeat bootstrapping in `bootstrapFixpoint` until reaching a fixpoint. In each iteration, we compile all meta-language definition into meta-language products. If the new baseline is equal to the baseline from the previous iteration, we have reached a fixpoint and return the new baseline.

To compare the language products of a baseline, we compare the name, version, artifacts, and generators of products. To compare generators, we need to compare the executables of generators (not modeled in Haskell). In practice, this boils down to comparing binary files byte-for-byte, ignoring nondeterministic metadata such as the creation date or the last modified date. We change meta-language versions each iteration in fig. 5.1 for illustrative purposes. However, `bootstrap` only changes versions once, to prevent baseline comparison from always failing because of version differences.

Bootstrapping fails with a dynamic exception if any compile operation fails. Otherwise, our algorithm soundly produces a new baseline. In the next section, we explain how to manage baselines in interactive environments.

5.4 Interactive Bootstrapping

A language workbench provides an interactive environment in which a language engineer can import language definitions, make changes to the defini-

tions in interactive editors, compile them into a language products, and test the changed languages. This allows a language engineer to quickly iterate over language design and implementation. Likewise, a meta-language engineer wants to quickly iterate over meta-language design and implementation [88]. Therefore, we need to support running bootstrapping operations in the interactive language workbench environment.

A language workbench manages an interactive environment with a *language registry* that manages all loaded language definitions and language products. The language registry loads language definitions and products *dynamically*, that is, while the environment is running without restarting the environment or starting a new one. The language workbench should react to loading, reloading, and unloading of language definitions and products, for example, by setting up file associations and updating editors.

To support interactive development, meta-language compilation interacts with the language registry. Instead of receiving a baseline as argument, in an interactive environment function `compile` from the previous section uses the language registry to retrieve language products. Thus, a change to the registry affects subsequent compilations.

Since bootstrapping relies on `compile`, in an interactive environment bootstrapping also interacts with the language registry. Instead of receiving a baseline as argument, in an interactive environment function `bootstrap` from the previous section uses the language registry to retrieve an initial baseline. Before calling `compile` in each iteration, `bootstrap` needs to load/reload the compiled products of version `v`. If bootstrapping succeeds, the new baseline stays in the language registry upon termination of `bootstrap`. But if bootstrapping fails with an exception, subsequent operations may not use the intermediate language products. To this end, `bootstrap` needs to rollback changes to the registry by unloading the language products of version `v`, and rolling back version changes in definitions.

Based on these changes to the algorithms and the language registry, our bootstrapping model supports interactive environments. Specifically, we can start a bootstrapping attempt with `bootstrap`, load a new baseline into the registry, and rollback the registry after bootstrapping failed or was canceled by the user.

5.5 Bootstrapping Breaking Changes

In the context of bootstrapping, a breaking change is a change to meta-language definitions such that fixpoint bootstrapping fails. Instead of treating such change as a whole, a breaking change needs to be decomposed into multiple smaller changes for which fixpoint bootstrapping succeeds.

For example, changing a keyword in the SDF meta-language is a breaking change, because it will cause parse failures for SDF source files elsewhere. Changing a keyword and all usages of the old keyword is also a breaking change, because we use the old baseline on the first build, which does not support the new keyword.

To perform such a breaking change, we need decompose it into *smaller non-*

breaking changes. Using such decomposition, fixpoint bootstrapping succeeds after each change. However, it is actually sufficient to only perform a full fixpoint bootstrap after the final change and to only find defects then. For the intermediate changes, it is enough to bootstrap a single iteration in order to construct a new baseline for the subsequent builds. To support this, we propose to extend the interactive environment with an additional bootstrapping operation that bootstraps a single iteration only. This will still find all defects in the final fixpoint bootstrapping, but intermediate defects may go unnoticed until then.

A common breaking change that occurs when evolving a meta-language is the change of a feature F to F'. For example, this includes changing the syntax of Stratego or changing the `Sig-gen` generator in SDF. We can decompose a change of feature F to F' in `M` by (1) adding F' as an alternative to F in `M`, (2) executing a single bootstrap iteration, (3) changing all source artifacts written in `M` to use F' instead of F, (4) executing a single bootstrap iteration, (5) removing F from `M`, and finally (6) performing a fixpoint bootstrap.

We have successfully used this decomposition for changing features in our evaluation.

5.6 Evaluation

To evaluate our bootstrapping method, we realized it in the Spoofax language workbench and bootstrapped Spoofax's eight meta-languages.

5.6.1 Implementation

We have implemented the model, general compilation algorithm, and general bootstrapping algorithm of our sound bootstrapping method in the interactive Eclipse environment of the Spoofax language workbench. With our implementation, a meta-language engineer can import meta-language definitions into Eclipse, make changes to the definitions, and run bootstrapping operations on the definitions to produce new baselines. The Eclipse console displays information about the bootstrapping process, e.g. when a new iteration starts, which artifacts were different during language product comparison, and any errors that occur during bootstrapping.

When bootstrapping fails, changes are reverted, and the console shows observed errors. Bootstrapping can also be cancelled by cancelling the bootstrapping job. When bootstrapping succeeds, the new baseline and meta-language definitions are dynamically loaded, such that the meta-language engineer can start making changes to the definitions and run new bootstrapping operations.

5.6.2 Meta-languages

To evaluate the bootstrapping method and implementation, we bootstrap Spoofax's meta-languages. Spoofax currently consists of eight meta-languages: SDF2, SDF3, Stratego, ESV, NaBL, TS, NaBL2, and DynSem. The generator dependencies between these meta-languages are shown in fig. 5.2.

Syntax used to be specified in the SDF2 [153] language, but we have since

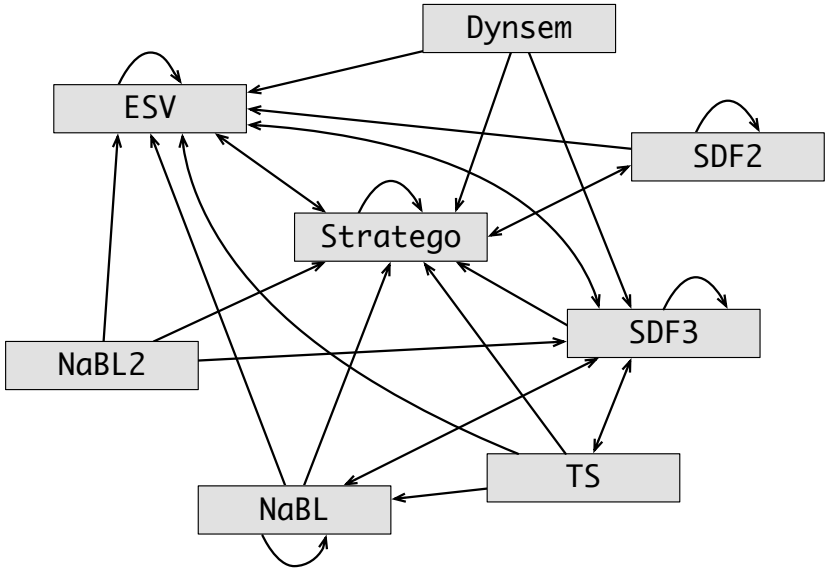


Figure 5.2: Generator dependencies between meta-languages. A generator dependency indicates that a meta-language requires (some) generators of a meta-language.

moved on to the more advanced SDF3 [158] language with syntax templates from which pretty-printers are automatically derived. SDF2 still exists because of compatibility reasons (some languages still use it) but also to use it as a target for generation. The SDF3 compiler generates SDF2, which the SDF2 compiler turns into a parse table. The SDF that we have been using as a running example in this paper is actually SDF3.

ESV is a domain-specific meta-language for specifying editor services such as syntax coloring, outlines, and folding. Every meta-language (including ESV itself) uses ESV to specify its editor services, such that Spoofax can derive an editor for the meta-languages.

Stratego [150, 18] is used for specifying term transformations and static semantics in several (meta)languages, and is also a common target for generation. For the name and type analysis domains, NaBL [89] is a domain-specific meta-language for specifying name analysis, and TS for specifying type analysis in terms of typing rules. NaBL2 is an evolution of NaBL that combines NaBL and TS in one language. Again, the older version of the language is kept for compatibility reasons. Finally, DynSem [147] is a meta-language for dynamic semantics specification through operational semantics.

We have successfully bootstrapped these meta-languages with our bootstrapping implementation.

5.6.3 Bootstrapping Changes

We evaluate our bootstrapping method and its implementation in Spoofax by bootstrapping changes to Spoofax's meta-languages. We could not test our bootstrapping implementation against existing changes made to the meta-languages, because our bootstrapping implementation expects meta-languages to be in a specific format which existing meta-languages are not. Therefore, we converted the meta-languages to this format and constructed realistic and interesting changes for evaluation.

We have logged the changes in the form of a Git repository[1], which contains a readme file explaining how to view the repository. Each change is tagged with a version in the Git repository. For each tag, sources and binaries of the created baseline are available. Tags for fixpoint bootstrapping operation also include the bootstrapping log. We now go over each change scenario to explain what we changed, why we made the change, and any issues that occurred.

Initial Bootstrap. To be able to bootstrap Spoofax's meta-languages, we convert the meta-language definitions to work with our bootstrapping implementation. We successfully bootstrap the meta-languages by running a fixpoint bootstrapping operation. Version *v2.1.0* is the first fixpoint bootstrap that includes all meta-languages, and will be used as a baseline for the next change.

SDF2 in SDF3. SDF2 currently exports a handwritten pretty-printer, which is imported by SDF3 to pretty-print SDF2 source files. A handwritten pretty-printer is bad for maintenance because it must be manually changed whenever the syntax changes. However, a generated pretty-printer is automatically updated in conjunction with changes to the syntax, which reduces the maintenance effort. Therefore, we convert SDF2's syntax to SDF3, such that SDF3's pretty-printer generator, generates a pretty-printer to replace the handwritten one.

This is a breaking change because SDF3 imports SDF2's pretty printer, and we change the pretty-printer that SDF2 exports, so SDF3's imports need to change. We decompose the change into three parts by applying the feature change decomposition shown previously: (1) we convert SDF2's syntax to SDF3 and export the generated pretty-printer, while still exporting the handwritten pretty-printer, (2) we change SDF3 to use the generated pretty-printer from SDF2, and (3) we remove the handwritten pretty-printer from SDF2. We apply each bootstrapping operation in the same environment, to test the interactive language workbench environment.

We first convert SDF2's syntax to SDF3, and run a single iteration bootstrapping operation to produce baseline *v2.1.1*. However, we have converted the syntax of SDF2 wrongly, constructor names are supposed to be in lowercase to retain compatibility with existing SDF2 transformations. Furthermore, lowercase constructor names such as `module` conflict with Stratego's syntax, which uses `module` as a reserved keyword. Therefore, we change SDF3 to support quotation marks in constructor names to support `'module`, which does not

[1] `https://github.com/spoofax-bootstrapping/bootstrapping`

conflict with Stratego, and run a single iteration bootstrapping operation to produce baseline *v2.1.2*.

We convert SDF2's grammar again, with lowercase constructor names, and prefix reserved keywords with ' where needed, run a single iteration bootstrapping operation to create baseline *v2.1.3*. We use the newly generated signatures and pretty-printer from SDF2 in SDF3, run a fixpoint bootstrapping operation (to confirm that the pretty-printer works), which succeeds, and produces baseline *v2.1.4*.

Finally, we clean up SDF2 by removing the handwritten pretty-printer. We also fix a bug in Stratego that causes some imports to loop infinitely during analysis, that was uncovered by the import dependencies between SDF2 and SDF3, which now mutually import each other. A fixpoint bootstrapping operation produces baseline *v2.1.5*.

Stratego in SDF3. We convert Stratego's syntax from SDF2 to SDF3 to also benefit from generated signatures and pretty-printers, instead of handwritten ones. This breaking change is decomposed in a similar way. However, we do not remove the handwritten pretty-printer yet, because multiple other meta-languages are using it, while we only change NaBL to use the generated one.

We convert Stratego's syntax to SDF3, run a single iteration bootstrapping operation to produce baseline *v2.1.6*. We change NaBL to use the newly generated Stratego signatures and pretty-printer, and run a fixpoint bootstrapping operation to produce baseline *v2.1.7*.

Results. We were able to successfully bootstrap eight meta-languages with realistic changes, including complex breaking changes that required multiple bootstrapping steps. Bootstrapping is sound because it terminates, finds defects when we introduce them, and produces a baseline when bootstrapping succeeds. We were also able to run multiple bootstrapping operations in the interactive language workbench environment, where the baseline produced from bootstrapping is loaded into the environment and used to kickstart the next bootstrapping operation.

5.7 Related Work

We now discuss related work on bootstrapping.

5.7.1 Bootstrapped General-Purpose Languages

Most general-purpose programming languages are bootstrapped. We discuss early languages and compilers that were bootstrapped.

The first programming language to be bootstrapped in 1959 is the NELIAC [66] dialect of the ALGOL 58 language. The main advantage of bootstrapping the compiler is implementation in a higher-level language. Instead of writing the compiler in assembly, it could be written in NELIAC itself, which is a much higher-level language than assembly. This allowed the compiler to be more easily be cross-compiled to the assembly of other machines, since the cross-compiled versions could be written in NELIAC.

Lisp was bootstrapped by creating a Lisp compiler written in Lisp, which was interpreted by an existing Lisp interpreter [142]. It is the first compiler that compiled itself by being interpreted by an existing interpreter of that language.

The Pascal P compiler [110] is a compiler for the minimal subset of standard PASCAL that it can still compile itself. It generates object code for a hypothetical stack computer SC. The first version of the compiler is written in assembly for SC. Using an assembler or interpreter for SC, the first compiler is compiled or executed. The compiler is bootstrapped by writing the compiler in itself, and compiling it with the existing compiler. The bootstrapped compiler is validated by comparing the assembled compiler binary and the bootstrapped compiler binary, a convention that we still apply to this day.

5.7.2 Bootstrapping

We now look at literature on the art of bootstrapping itself.

Tombstone diagrams (also called T-diagrams) are a graphical notation for reasoning about translators (compilers), first introduced in [16] and extended with interpreters and machines in [31]. T-diagrams are most commonly used to describe bootstrapping, cross-compilation, and other processes that require executing complex chains of compilers, interpreters, and machines [95]. T-diagrams are a useful tool for graphically reasoning about compilers and bootstrapping, orthogonal to the bootstrapping framework presented in this paper.

Axiomatic bootstrapping [8] is an approach for reasoning about bootstrapping using axioms and equations between those axioms, to verify if a change to a compiler will result in a successful bootstrap. They present axioms for an interactive ML runtime and compiler, which compiles to native code, and needs to deal with multiple architectures, calling conventions, and binary formats. For example, instantiating the axioms and equations show that changing the calling convention of the compiler causes bootstrapping to fail, and that a special cross-compilation operation can bootstrap the change.

Axioms are a useful tool to verify if a single bootstrapping iteration will work, and to reason about how a breaking change should be split up over multiple bootstrapping steps. However, axioms cannot be used to find hidden defects, such as the example from fig. 5.1, because those defects can manifest over an arbitrary number of iterations. Axioms also cannot be used to reason if a fixed point can be reached, for the same reason. Therefore, it is complementary to the bootstrapping framework presented in this paper.

5.7.3 Language Workbenches

We now look at related work on bootstrapping in language workbenches.

Xtext [12] is a framework and IDE for the development of programming languages and DSLs. Its Grammar and Xtend meta-languages are bootstrapped. However, Xtext does not support dynamic loading, so it is not possible to bootstrap the meta-languages in the language workbench environment.

MPS [159] is a projectional language workbench. It has several meta-languages which are bootstrapped, but are read-only in the language work-

bench environment, meaning that they cannot be bootstrapped in the language workbench environment. MPS supports dependencies between languages through its powerful extension system. A meta-language can be extended, and that extension could be bootstrapped. However, MPS does not support versioning or undoing changes, making rollbacks impossible if defects are introduced.

MetaEdit+ [78] is a graphical language workbench for domain-specific modeling. Some of its meta-languages are bootstrapped, and can be bootstrapped in the language workbench environment. When changes are applied, they immediately apply to the bootstrapped meta-language and other languages. If an applied change breaks the language, the change can be undone or abandoned entirely to go back to a previous working state, after which the error can be fixed. However, they do not document their bootstrapping method, so it cannot be applied to other language workbenches.

Ensō [97, 138] is a project to enable a software development paradigm based on interpretation and integration of executable specification languages. Ensō's meta-languages are bootstrapped, but has no general framework for fixpoint bootstrapping or versioning of meta-languages. The meta-language engineer has to write code which handles fixpoint bootstrapping and versioning specifically for their meta-languages.

Rascal [81] is a metaprogramming language and IDE for source code analysis and transformation. In the current version, Rascal's parser is bootstrapped, but the rest is implemented as an interpreter in Java. Development versions include a new Rascal compiler which is completely bootstrapped. However, the Rascal IDE has no general support for fixpoint bootstrapping or versioning of languages.

SugarJ [37, 35] is a Java-based extensible programming language that allows programmers to extend the base language with custom language features. In principle, SugarJ can be bootstrapped, because its compiler is written in Java and Java is a subset of SugarJ. However, in practice this was never done, and it is not obvious if that would actually work.

Racket [146] is an extensible programming language in the Lisp/Scheme family, which can serve as a platform for language creation, design, and implementation. DrRacket [41, 42] is the Racket IDE. Racket is mostly bootstrapped, but the core of the compiler and interpreter are implemented in C. The parts of Racket that are written in Racket can be changed interactively in DrRacket, which affects subsequently running Racket programs. A defect introduced in Racket's self definition may prevent bootstrapping to succeed, which requires a restart of the DrRacket IDE.

5.7.4 Staged Metaprogramming

Staged metaprogramming approaches such as MetaML [143], MetaOCaml [21], Mint [162], and LMS [125] provide typesafe run-time code generation, which ensures that generated code does not contain typing defects. However, these approaches do not provide support for bootstrapping.

5.8 Conclusion

Bootstrapping is an efficient means for detecting defects in compiler implementations and should be useful for language workbenches as well. However, bootstrapping compiler-compilers of language workbenches needs to handle the intricate interactions between meta-languages. Unfortunately, previous literature on bootstrapping ignores these intricacies.

We present a sound method for meta-language bootstrapping. Given a baseline and updated meta-language definitions, our bootstrapping algorithm constructs a new baseline through fixpoint self-application of the meta-languages. We explain how our algorithms can be used in interactive environments and how to decompose breaking changes that occur when evolving meta-languages.

We have implemented the approach in the Spoofax language workbench and evaluated it by successfully bootstrapping eight interdependent meta-languages, and report on our experience with bootstrapping two breaking changes. This makes Spoofax into a laboratory for meta-language design experimentation.

Acknowledgements

This research was supported by NWO/EW Free Competition Project 612.001.114 (Deep Integration of Domain-Specific Languages) and NWO VICI Project (639.023.206) (Language Designer's Workbench).

Reflection: Language Workbench Pipelines

6

Our work on bootstrapping enables systematic bootstrapping of the meta-languages of language workbenches in an interactive metaprogramming system. However, while working on bootstrapping, we noticed that language workbenches implement a very complicated *pipeline*. A language workbench pipeline builds (meta-)language specifications into (meta-)language products; parses, analyses, and transforms programs using these products; and provides feedback to programmers through editor services such as inline error messages, code styling, structure outlines, and code completion. Such a pipeline roughly flows as follows:

- Meta-language specifications are bootstrapped using a baseline, to produce new meta-language products.
- Meta-language programs (which specify a language) go through the pipeline: parse, analyze, transform, and provide feedback to the language developer, using meta-language products.
- Language specifications are built into language products.
- Programs go through the pipeline and provide feedback to programmers, using language products.
- When programs are transformed or compiled into other programs that require more processing or feedback, the pipeline continues.

To make an interactive (meta)programming system based on such a pipeline, it must be incremental, scalable, and easy to develop and maintain. While the pipeline was improved a lot with our work on Spoofax Core and bootstrapping, it still suffered from several problems which we now analyze.

Problem Analysis

The pipeline of Spoofax overspecifies certain dependencies, causing a loss of incrementality. For example, when we change the name and type analysis specification of a language, rebuild the language specification into a language product, and then edit a program of that language, Spoofax will reparse the program even though the parser did not change. This is because Spoofax does not perform fine-grained dependency tracking, because the development effort to do so would be large, requiring incrementality techniques such as caching, cache invalidation, dependency tracking, and change detection.

Conversely, Spoofax's pipeline also underspecifies certain dependencies, causing loss of correctness. For example, when we change the NaBL meta-language, and then edit a language specification that uses NaBL, Spoofax will not automatically rebuild the NaBL meta-language, resulting in inconsistent behavior. The language developer must first manually build (bootstrap)

NaBL, and then build their own language. This is again because Spoofax does not perform fine-grained dependency tracking, but also because eagerly rebootstrapping the meta-languages for every change would take a lot of time.

In some cases, Spoofax's pipeline *is* incremental and correct. For example, the separate transformation (compilation) of the source files of the SDF meta-language into normalized SDF files for parse table generation, pretty-printer rules, and syntactic code completion rules, is incremental and correct. Separate compilation is simpler to implement because there are no dependencies between source files. However, to achieve this, we still manually implement an incremental pipeline with caching, invalidation, and change detection.

Finally, Spoofax's pipeline is implemented in five different formalisms:

1. A custom build system that parses, analyzes, and transforms programs, which is incremental only if the transformation is a separate transformation (i.e., does not depend on other files or modules).
2. A language registry which (re)loads languages dynamically, enabling live language development.
3. Maven POM files which instruct Maven to download Java dependencies and compile Java code.
4. The meta-language bootstrapping system from the previous chapter.
5. An incremental build system for building language specifications, based on Pluto [36], a sound and optimal incremental build system with dynamic dependencies.

Because the pipeline is specified in five different formalisms, it is harder to understand and change, and several opportunities for incrementality are lost because different formalisms do not properly communicate.

Vision

Given this problem analysis, and the desire to develop correct and responsive interactive programming systems for language workbench pipelines, we need a systematic method for developing these pipelines. This method should have a single formalism in which pipelines can be concisely and directly expressed, without the pipeline developer having to explicitly think about incrementality. Pipelines specified in this formalism are correctly and incrementally executed, and scale down to low-impact changes and large inputs.

In the next chapter (chapter 7), we show PIE, a formalism for describing pipelines, and a runtime for correctly and incrementally executing these pipelines. PIE reuses the incremental build algorithm from Pluto [36]. However, Pluto's incremental build algorithm does not scale because it needs to traverse the entire dependency graph after each change, which becomes slow with many low-impact changes and large dependency graphs. In chapter 8 we solve this scalability problem with a new change-driven incremental build algorithm.

PIE: A DSL, API, and Runtime for Interactive Software Development Pipelines

7

Abstract

Context. Software development pipelines automate essential parts of the software engineering processes, such as compiling and continuous integration testing. In particular, *interactive* pipelines, which process events in a live environment such as an IDE, require *responsive* results for low-latency feedback, and *persistence* to retain low-latency feedback between restarts.

Inquiry. Developing an incrementalized and persistent version of a pipeline is one way to improve responsiveness, but requires implementation of dependency tracking, cache invalidation, and other complicated and error-prone techniques. Therefore, interactivity complicates pipeline development if responsiveness and persistency become responsibilities of the pipeline programmer, rather than being supported by the underlying system. Systems for programming incremental pipelines exist, but do not focus on ease of development, requiring a high degree of boilerplate, increasing development and maintenance effort.

Approach. We develop PIE, a DSL, API, and runtime for developing interactive software development pipelines, where ease of development *is* a focus. The PIE DSL is a statically typed and lexically scoped language. PIE programs are compiled to programs implementing the API, which the PIE runtime executes in an incremental and persistent way.

Knowledge. PIE provides a straightforward programming model that enables direct and concise expression of pipelines without boilerplate, reducing the development and maintenance effort of pipelines. Compiled pipeline programs can be embedded into interactive environments such as code editors and IDEs, enabling timely feedback at a low cost.

Grounding. Compared to the state of the art, PIE reduces the code required to express an interactive pipeline by a factor of 6 in a case study on syntax-aware editors. Furthermore, we evaluate PIE in two case studies of complex interactive software development scenarios, demonstrating that PIE can handle complex interactive pipelines in a straightforward and concise way.

Importance. Interactive pipelines are complicated software artifacts that power many important systems such as continuous feedback cycles in IDEs and code editors, and live language development in language workbenches. New pipelines, and evolution of existing pipelines, is frequently necessary. Therefore, a system for easily developing and maintaining interactive pipelines, such as PIE, is important.

7.1 Introduction

A pipeline is a directed acyclic graph of processors in which data flows from the output of one processor to the input of its succeeding processors. Pipelines are ubiquitously used in computer hardware and software. E.g., in hardware, CPUs contain instruction pipelines that allow interleaved execution of multiple instructions that are split into fixed stages. Software pipelines compose software components by programmatically connecting their input and output ports (e.g., UNIX pipes).

In software development, pipelines are used to automate parts of the software engineering process, such as building software systems via build scripts, or continuously testing and integrating the composition of subsystems. Such pipelines are suitable for batch-processing, and often run isolated on remote servers without user interaction.

Interactive software development pipelines build software artifacts, but react instantly to changes in input data and provide timely feedback to the user. Typical examples are continuous editing of source code in an IDE, providing feedback through editor services such as syntax highlighting; selective re-execution of failing test cases in the interactive mode of a build system during development; or development of languages in a language workbench [39].

Interactive pipelines focus on delivering *timely* results when processing an event, such that the user can subsequently act on the results. Furthermore, an interactive software development pipeline should *persist* its state on non-volatile memory so that a session can be restarted without re-execution. Especially in the context of an IDE, restarting the development environment should not trigger re-execution of the entire pipeline, especially if pipeline steps are costly, such as advanced static analyses [141].

Interactivity complicates the development of pipelines, if *timeliness* and *persistency* become responsibilities of the pipeline programmer, rather than being supported by the underlying system. Developing an incrementalized version of an expensive operation is one way to reduce the turnaround time when re-executing the operation. However, implementing support for incrementality in a pipeline is typically complicated and error-prone. Similarly, persisting the result of expensive operations reduces the turnaround time when restarting a session, but requires tedious management of files or a database. Furthermore, when persistency is combined with incrementality, dependency tracking and invalidation is required, which is also complicated and error-prone. Therefore, an expressive system for easily developing correct incremental and persistent interactive software development pipelines is required.

One system that partially achieves this is Pluto [36], a sound and optimal incremental build system. Pluto supports dynamic dependencies, meaning that dependencies to files and other build steps are created during build execution (as opposed to before or after building), enabling both increased incrementality through finer-grained dependencies, and increased expressiveness. While Pluto focusses on build systems, it is well suited for expressing correct incremental and persistent pipelines. However, ease of development is not a focus of Pluto, as pipelines are implemented as Java classes, requiring significant

boilerplate which leads to an increase in development and maintenance effort. Furthermore, persistence in Pluto is not fully automated because pipeline developers need to manually thread objects through pipelines to prevent hidden dependencies, and domain-specific features such as file operations are not first class. These are open problems that we would like to address.

In this paper, we introduce PIE, a DSL, API, and runtime for programming interactive software development pipelines, where ease of development *is* a focus. The PIE DSL provides a straightforward programming model that enables direct and concise expression of pipelines, without the boilerplate of encoding incrementality and persistence in a general-purpose language, reducing development and maintenance effort. The PIE compiler transforms high-level pipeline programs into programs implementing the PIE API, resulting in pipeline programs that can be incrementally executed and persisted to non-volatile memory to survive restarts with the PIE runtime. Compiled pipeline programs can be embedded in an interactive environment such as an IDE, combining coarse grained build operations with fine-grained event processing. To summarize, the paper makes the following contributions:

- The PIE language, a DSL with high-level abstractions for developing interactive software development pipelines without boilerplate.
- The PIE API for implementing foreign pipeline functions, and as a compilation target for the DSL, with reduced boilerplate.
- The PIE runtime that executes pipelines implemented in the API in an incremental and persistent way, which fully automates persistence and automatically infers hidden dependencies.
- An evaluation of PIE in two critical case studies: (1) modeling of the pipeline of a language workbench in an IDE setting, and (2) a pipeline for incremental performance testing.

The PIE implementation is available as open source software [84].

Outline. The paper continues as follows. In section 7.2 we describe requirements for interactive software development pipelines, review the state of the art, and list open problems. In section 7.3 we illustrate PIE by example. In section 7.4 we describe the PIE API and runtime. In section 7.5 we describe the syntax, static semantics, and compilation of the PIE DSL in more detail. In sections 7.6 and 7.7 we present critical case studies of the application of PIE in an interactive language workbench and an interactive benchmarking setting. In section 7.8 we discuss related work. In section 7.9 we discuss directions for future work. Finally, we conclude in section 7.10.

7.2 Problem Analysis

In this section, we first describe requirements for interactive software development pipelines, review the state of the art, and list open problems.

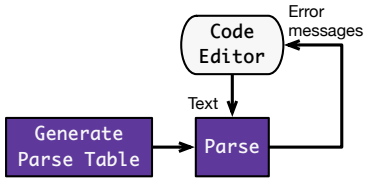


Figure 7.1: Example of an interactive software development pipeline, where text from a code editor is parsed, and parse error messages are displayed in the editor.

7.2.1 Requirements

We first describe the requirements for interactive software development pipelines. In order to do so, we use the example pipeline from fig. 7.1 as the running example in this section. In this pipeline, a code editor parses its text buffers in order to display error messages interactively to the programmer. Parsing requires a parse table, which is generated by an external process and may change when a new version of a language is deployed, with new syntax that requires regeneration of the parse table. We identify the following requirements for interactive software development pipelines:

Incrementality. A pipeline should attempt to recompute only what has been affected by a change. For example, when only a text buffer in the code editor changes, the pipeline reparses the text and new error messages are displayed, but the generated parse table is reused because it did not change.

Correctness. Incremental pipeline executions must have the same results as from-scratch batch executions. For example, if the parse table *does* change, the pipeline also reparses text and displays new error messages.

Persistence. Results of computation should be persisted to disk in order to enable incrementality after a restart of the pipeline. For example, if we restart the code editor, the parse table is retrieved from disk instead of requiring a lengthy recomputation.

Expressiveness. In practice, pipelines are a lot more complex than the simple example shown here. It should be possible to express more complex pipelines as well.

Ease of development. Pipelines are complex pieces of software, especially when the previous requirements are involved. Therefore, the development and maintenance effort of pipelines should be low.

7.2.2 State of the Art

We now review the state of the art in interactive software development pipelines, and determine to what extent existing tools meet the requirements, focussing on build systems.

Make [133], and systems with similar dependency management (e.g., Ninja, SCons, MSBuild, CloudMake, Ant), are tools for developing build systems based on declarative rules operating on files. These tools support incremental

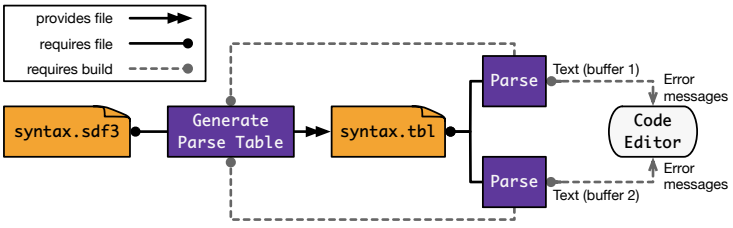


Figure 7.2: Pluto dependency graph created by executing the pipeline from fig. 7.1, where the code editor has 2 open text buffers.

builds, but incrementality is limited to static file dependencies which are specified up front in the build rules. Because dependencies cannot be the result of computation, the dependencies must either be soundly overapproximated, which limits incrementality, or underapproximated, which is unsound. For example, a Makefile that determines the version of a Java source file, in order to parse it with the corresponding parse table file, must depend on all parse table files instead of a single one. Therefore, a system that supports a more expressive dependency mechanism is required. A detailed discussion of dependency expressiveness can be found in section 7.8, but in this section, we focus on the system with the highest dependency expressivity: Pluto.

Pluto [36] is a sound and optimal incremental build system with support for dynamic dependencies. A build system in Pluto is implemented in terms of *builders*, which are functions that perform arbitrary computations and dynamically record dependencies to files and other builders during execution. Executing a builder with an input produces a *build*, containing an output object and recorded dependencies.

Figure 7.2 illustrates the dependency graph Pluto produces when it executes the pipeline of fig. 7.1 where the code editor has two open text buffers. The dependency graph differs from the pipeline by containing builds (function calls) instead of builders (function definitions). For example, the pipeline has one parse builder, but two parse builds, one for each text buffer. We use this dependency graph to illustrate Pluto's adherence to requirements for interactive pipelines:

Incrementality. The code editor has two text buffers open, which have separate dependencies to a parse build. When one text buffer changes, only the corresponding parse build is recomputed. Therefore, Pluto supports fine-grained incrementality.

Correctness. The parse builds depend on the parse table build, such that when the parse table is regenerated, the parse builds are re-executed, and new error messages are displayed in the editor. Pluto enforces this by performing *hidden dependency detection*. That is, if a build requires a file, without requiring the build that provides that file, Pluto marks this as an error and aborts execution.

Persistence. While not shown in the dependency graph, builds are persisted

to disk to survive restarts.

Expressiveness. Dependencies are recorded during build execution, allowing builds to depend on files or call other builds, based on results of computation. For example, when parsing Java code, the parse builder may choose to depend on a different parse table, based on whether we want to parse text of version 8 or 9 of Java. This greatly increases the expressiveness required for interactive software development pipelines.

Ease of development. Pluto build systems are implemented in Java, requiring significant boilerplate.

To summarize, Pluto provides a great foundation for implementing interactive software development pipelines, but does not cater to the pipeline developer because ease of development is not a focus, leading to a higher implementation and maintenance effort than necessary.

7.2.3 Open Problems

The main problem is that Pluto build systems are not easy to develop. We list four concrete open problems.

Boilerplate. Pipelines in Pluto are written in Java, which has a rigid and verbose syntax, requiring significant *boilerplate*. Pipelines are implemented as classes extending the `Builder` abstract class, as seen in listing 7.1. Such a class requires generics for specifying the input and output type, a `factory` and constructor enabling other builders to create instances of this builder to execute it, a `persistentPath` method for persistence, and finally a `build` method that performs the actual build computation. The `Parse` builder requires an inner class for representing multiple input values, which must correctly implement `equals` and `hashCode`, which Pluto uses to detect if an input has changed for incrementality. Finally, calling other builders through `requireBuild` is verbose, because the `factory` is referenced, and the result is unwrapped with `.val`.

Semi-automated persistence. Pipeline developers are required to implement the `persistentPath` method of a builder and return a *unique* and *deterministic* filesystem path where the result of the builder and its input are persisted. It must be unique to prevent overlap with other builders or other inputs. For example, if the `Parse` builder persists results to the same file for different text buffers, it overwrites the persisted result of other builds. It must be deterministic such that the persisted file can later be found again. Since the OS filesystem is used for persistence, there are also limitations to which characters can be used in paths, and to how long a path can be. For example, on Windows, the current practical limit is 260 characters which is frequently reached with deeply nested paths, causing persistence to fail.

Hidden dependencies. Hidden dependency detection is crucial for sound incremental builds, but is also cumbersome. In the pipeline in listing 7.1, we must construct a build request object for the parse table generator, pass that object to the `Parse` builder, and require it to depend on the parse table generator. This becomes tedious especially in larger and more complicated pipelines.

```
class GenerateTable extends Builder<File, Out<File>> {
  static BuilderFactory<File, Out<File>, GenerateTable> factory =
    BuilderFactoryFactory.of(GenerateTable.class, File.class);

  GenerateTable(File syntaxFile) {
    super(syntaxFile);
  }
  @Override File persistentPath(File syntaxFile) {
    return new File("generate-table-" + hash(syntaxFile));
  }
  @Override Out<File> build(File syntaxFile) throws IOException {
    require(syntaxFile);
    File tblFile = generateTable(syntaxFile);
    provide(tblFile);
    return OutputPersisted.of(tblFile);
  }
}
class Parse extends Builder<Parse.Input, Out<ParseResult>> {
  static class Input implements Serializable {
    File tblFile; String text; BuildRequest tblReq;
    Input(File tblFile, String text, BuildRequest tblReq) {
      this.tblFile = tblFile;
      this.text = text;
      this.tblReq = tblReq;
    }
    boolean equals(Object o) {/* omitted */}
    int hashCode() {/* omitted */}
  }
  @Override Out<ParseResult> build(Input input) throws IOException {
    requireBuild(input.tblReq);
    require(input.tblFile);
    return OutputPersisted.of(parse(input.tblFile, input.text));
  }
  /* ... other required code omitted ... */
}
class UpdateEditor extends Builder<String, Out<ParseResult>> {
  @Override Out<ParseResult> build(String text) throws IOException {
    File syntaxFile = new File("syntax.sdf3");
    File tblFile = requireBuild(GenerateTable.factory, syntaxFile).val;
    BuildRequest tblReq = new BuildRequest(GenerateTable.factory, syntaxFile);
    return requireBuild(Parse.factory, new Parse.Input(tblReq, tblFile, text));
  }
  /* ... other required code omitted ... */
}
```

Listing 7.1: The parsing pipeline implemented as Java classes in Pluto.

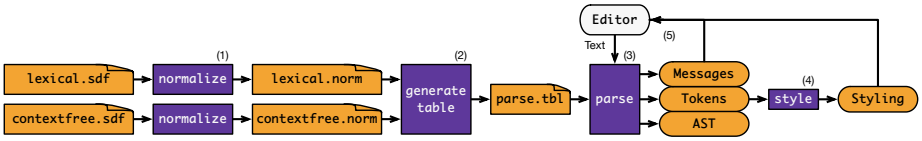


Figure 7.3: The example pipeline: (1) normalization of SDF syntax definition source modules, (2) generation of a parse table from the normalized modules comprising the definition for a language, (3) parsing the text of an editor using the parse table, (4) computing the styles for the parsed tokens, and (5) displaying the styling and error messages in an editor.

Missing domain-specific features. Finally, path (handle to file or directory) and list operations, which are prevalent in software development pipelines, are not first class in general-purpose languages such as Java.

Solving these concrete problems requires a proper abstraction over interactive software development pipelines, which we present in subsequent sections.

7.3 PIE by Example

To solve the open problems from the previous section, we introduce PIE: a DSL, API, and runtime for developing and executing interactive software development pipelines. Pipelines in PIE have minimal boilerplate, fully automated persistence, automatically infer hidden dependencies, and have domain-specific features such as path and list operations. In this section we illustrate PIE by means of an example that combines building and interaction. We discuss the example and the requirements for this pipeline, present the pipeline in the PIE DSL, and discuss its features and execution.

Example Pipeline: Syntax-Aware Editors. As example we consider a code editor with syntax styling based on a syntax definition. The pipeline to support this use case is depicted by the diagram in fig. 7.3. It generates a parse table from a syntax definition, parses the program text of an editor, computes syntax styling for each token, and finally applies the computed syntax styling to the text in the editor. We want this pipeline to be *interactive* by embedding it into the IDE such that changes to the syntax definition as well as changes to the text in an editor are reflected in updates to syntax styling. The example in fig. 7.3 is representative for *language workbenches* [38, 39], which support edits on a language definition that are immediately reflected in the programming environment that is derived from it.

Concretely, we instantiate the pipeline with components from the Spoofax language workbench [75]. We process an SDF [153, 158] syntax definition in two stages. First, syntax definition modules are separately transformed (normalized) to a core language. Next, the normalized modules comprising the syntax definition for a language are transformed to a parse table. The parse table is interpreted by a scannerless parser [152, 15] to parse the contents of an editor, returning an AST, token stream, and error messages. A syntax

```
func normalize(file: path, includeDirs: path*) -> path = {
  requires file; [requires dir with extension "sdf" | dir <- includeDirs];
  val normFile = file.replaceExtension("norm");
  val depFile = file.replaceExtension("dep");
  exec(["sdf2normalized"] + "$file" + ["-I$dir" | dir <- includeDirs] +
    "-o$normFile" + "-d$depFile");
  [requires dep by hash | dep <- extract-deps(depFile)];
  generates normFile; normFile
}
func extract-deps(depFile: path) -> path* = foreign
func generate-table(normFiles: path*, outputFile: path) -> path = {
  [requires file by hash | file <- normFiles];
  exec(["sdf2table"] + ["$file" | file <- normFiles] + "-o$outputFile");
  generates outputFile; outputFile
}
func exec(arguments: string*) -> (string, string) = foreign

data Ast = foreign {} data Token = foreign {} data Msg = foreign {}
data ParseTable = foreign {} data Styling = foreign {}
func table2object(text: string) -> ParseTable = foreign
func parse(text: string, table: ParseTable) -> (Ast, Token*, Msg*) = foreign
func style(tokenStream: Token*) -> Styling = foreign
func update-editor(text: string) -> (Styling, Msg*) = {
  val sdfFiles = [./lexical.sdf, ./contextfree.sdf];
  val normFiles = [normalize(file, [./include]) | file <- sdfFiles];
  val parseTableFile = generate-table(normFiles, ./parse.tbl);
  val (ast, tokenStream, msgs) = parse(text, table2object(read
   parseTableFile));
  (style(tokenStream), msgs)
}
```

Listing 7.2: PIE DSL program for the pipeline illustrated in fig. 7.3. Identifiers of foreign functions and data types are omitted for brevity.

highlighter annotates tokens in the token stream with styles.

Integrated Pipelines with the PIE DSL. Listing 7.2 shows the pipeline program in the PIE DSL. We first explain what each function does, and then discuss the features and execution of PIE in more detail.

The `normalize` function executes a command-line tool to normalize an SDF source file into a normalized version that is ready for parse table generation, and retrieves (dynamic) dependencies from the generated dependency (.dep) file, implemented by the `extract-deps` foreign function. The `generate-table` function executes a command-line tool on normalized files, creating a parse table file. The `parse` function, when given a parse table object, parses text into an AST, token stream, and error messages. The `style` function produces a styling based on a token stream, which can be used in source code editors for styling the text of the source code. Finally, the `update-editor` function defines the complete pipeline by composing all previously defined functions.

Composing Pipelines with Functions. In PIE, pipelines are defined in terms of function definitions which are the reusable processors of the pipeline, and function calls that compose these processors to form a pipeline. Function calls

register a dynamic *call dependency* from caller to callee.

Domain-Specific Types and Dependencies. Since build pipelines often interact with files and directories, PIE has native support for `path` types and several operations on paths. Path literals such as `./lexical.sdf` provide an easy way to instantiate relative or absolute paths. The `requires` operation dynamically registers a *path dependency* from the current function call to the path, indicating that the function reads the path, whereas `generates` records a dependency indicating the function creates or writes to the path. The `read` operation reads the text of a given path, and also registers a path dependency.

Path dependencies to directories can specify a filter such as `with extension "sdf"` to only create dependencies to files inside the directory that match the filter. Finally, path dependencies can specify how changes are detected. For example `requires dep by hash` indicates that a change is only detected when the hash of the file changes, instead of the (default) modification date, providing more fine grained dependency tracking.

Foreign Functions and Types. Some functions are `foreign`, indicating that they are implemented outside of the PIE DSL, either because they are outside of the scope of the DSL (e.g., text processing required for `extract-deps`), or because they require system calls. For example, `exec` is a foreign function that takes a list of command-line arguments, executes a process with those arguments, and returns its standard output and error text. Unlike `read`, `exec` is not first class, because it does not induce special (path) dependencies.

PIE contains several built-in types such as `string` and `bool`, but foreign types can be defined to interface with existing types. Foreign data types are required to integrate with existing code, such as an editor that expects objects of type `Styling` and `Msg`, returned by foreign functions `parse` and `style`.

Comprehensions. Pipelines frequently work with lists, which are natively supported in PIE by annotating a type with `*` multiplicity. Lists are instantiated with list literals between `[ ]`, and concatenated using `+`. List comprehensions such as `[f(elem) | elem <- elems]` transform a list into a new list by applying a function `f` to each element of the list.

Execution and IDE Integration. To execute a pipeline, we compile it into a program implementing the PIE API. We embed the compiled pipeline, together with the PIE runtime, into an IDE such as Eclipse. When an editor in Eclipse is opened or changed, it calls the `update-editor` function through the PIE runtime with the text from the editor. The PIE runtime then incrementally executes (and persists the results of) the pipeline and returns `Styling` and `Msg` objects, which Eclipse displays in the editor. Because the results of the pipeline are persisted, a restart of Eclipse does not require re-execution of the pipeline. This becomes especially important with larger pipelines.

Solutions to Open Problems. PIE solves the open problems listed in section 7.2. First of all, PIE minimizes boilerplate by enabling direct expression of pipelines in the PIE DSL through function definitions, function calls, foreign data types and functions, and path dependencies. The compiler of the DSL generates

the corresponding boilerplate. Furthermore, PIE supports fully automated persistence. There is no need to specify where the result of a function call is stored. The PIE runtime stores results automatically based on the function name and input arguments. It persists the function arguments, return value, and dependencies in a key-value store, preventing filesystem issues.

Hidden call dependencies are automatically inferred. In other words, when a function requires files that are generated by another function, the first function does not explicitly need to call the latter. For example, `generate-table` `requires` files that `normalize` `generates`, but does not need to explicitly call `normalize` to record a call dependency which keeps the required files up-to-date. The PIE runtime infers these dependencies by keeping track of which function call generated a file, further reducing boilerplate. Note that this only infers *call dependencies*, not path dependencies, which still need to be declared by the pipeline programmer.

Finally, the PIE DSL caters to the pipeline developer by including domain-specific features – such as path type and operations, list type and comprehensions, string and path interpolation, and tuples – to make pipeline development convenient.

Solving these problems reduces the implementation and maintenance effort. The equivalent Pluto implementation for this pipeline requires 396 lines of Java code in 8 files (excluding comments and newlines), whereas the PIE implementation is over 6 times shorter by only requiring 62 lines of code in 2 files. The PIE code consists of 34 lines of PIE DSL code, and 28 lines of PIE API code for interfacing with foreign functions.

7.4 PIE API and Runtime

In this section, we review the PIE API and runtime, and our reasons for not directly reusing the Pluto runtime.

7.4.1 API

The PIE API is a Kotlin [71] library for implementing PIE function definitions on the JVM. Kotlin is a programming language with a focus on reducing verbosity and increasing extensibility compared to Java, while maintaining fully compatible with Java by running on the JVM. It shares many goals with Scala [34], but additionally focusses on fast compile times and simplicity. We chose to specify the API in Kotlin instead of Java, because it has a more flexible and concise syntax. The PIE API is heavily based on the Pluto API, but uses terminology from the pipeline domain (functions instead of builders), and requires less boilerplate.

Listing 7.3 illustrates the parsing pipeline implemented in the PIE API. A pipeline function definition is implemented by creating a class which subtypes the `Func` interface and overrides the `exec` function. The `exec` function takes an input, is executed in an execution context `ExecContext`, and produces an output. The execution context enables calling other pipeline functions through the `requireCall` function, and recording of path dependencies through the `require` and `generate` functions, using Kotlin's extension functions to make

```
typealias In = Serializable
typealias Out = Serializable
interface Func<in I:In, out O:Out> {
  fun ExecContext.exec(input: I): O
}
interface ExecContext {
  fun <I:In, O:Out, F:Func<I, O>> requireCall(clazz: KClass<F>, input: I,
    stamper: OutputStamper = OutputStampers.equals): O
  fun require(path: PPath, stamper: PathStamper = PathStampers.modified)
  fun generate(path: PPath, stamper: PathStamper = PathStampers.hash)
}

class GenerateTable: Func<PPath, PPath> {
  override fun ExecContext.exec(syntaxFile: PPath): PPath {
    require(syntaxFile)
    val tableFile = generateTable(syntaxFile)
    generate(tableFile)
    return tableFile
  }
}
class Parse: Func<Parse.Input, ParseResult> {
  data class Input(val tableFile: PPath, val text: String): Serializable
  override fun ExecContext.exec(input: Input): ParseResult {
    require(input.tableFile)
    return parse(input.tableFile, input.text)
  }
}
class UpdateEditor: Func<String, ParseResult> {
  override fun ExecContext.exec(text: String): ParseResult {
    val tableFile = requireCall(GenerateTable::class, path("syntax.sdf3"))
    return requireCall(Parse::class, Parse.Input(tableFile, text))
  }
}
```

Listing 7.3: The PIE `Func` API and implementation of a parsing pipeline in that API.

these functions accessible without a qualifier. The PIE runtime uses this execution context for dependency tracking and hidden dependency inference.

Inputs of `Func` implementations must be immutable, `Serializable`, and have an `equals` and `hashCode` implementation. These properties are required so that the PIE runtime can assume objects do not change inside a cache, can persist objects to non-volatile memory, and can detect if an object has changed for incrementality. The types used in listing 7.3 all adhere to these properties. Furthermore, Kotlin's data classes automatically implement `equals` and `hashCode`, reducing boilerplate for multiple input arguments.

Outputs of functions must adhere to the same properties, with the exception that outputs can opt-out of serialization. Some outputs are in-memory object representations and cannot be serialized, are too large to be serialized, or are not immutable. PIE supports these kind of objects as outputs of function calls, by wrapping the output in a special class (`OutTransient`) which prevents serialization. PIE still caches these outputs in volatile memory. However, when the runtime is restarted (thus clearing the in-memory cache), and such an

output is requested by calling the function, PIE re-executes the function to recreate the output.

Although it is possible to implement a full pipeline directly in this API, there is more boilerplate involved compared to writing the pipeline in the PIE DSL. Therefore, the API should only be used for implementing foreign functions, such as interfacing with a parse table generator and parser, or for executing system calls such as executing command-line tools. However, reduced boilerplate for implementing foreign functions reduces implementation and maintenance effort.

7.4.2 Runtime

The job of the PIE runtime is to execute a pipeline – represented as a set of `Func` implementations, from compiled PIE DSL code, and from foreign function implementations against the PIE API – in an incremental and persistent way. The runtime is largely based on the Pluto runtime, from which we inherit the sound and optimal incremental and persistent build algorithm. However, we incorporate fully automated persistence and hidden dependency inference in the PIE runtime.

The runtime calls a `Func` by calling its `exec` function with an input argument, under an execution context. During execution, a function may call other functions, and record path dependencies, through the execution context, and finally return a value. After a function has been executed, the runtime persists the returned value and recorded dependency information in a key-value store, by mapping the function call (`Func` instance and input argument) to the returned value and dependency information. This mapping is used by the incremental build algorithm as a cache and for retrieving dependency information. We use the LMDB [140] key-value database, which persists to a single file on the filesystem, and is memory-mapped for fast read access. Therefore, we fully automate persistence, meaning that pipeline developers are freed from reasoning about persistence.

To infer hidden dependencies, whenever a path (handle to file or directory) is generated, the runtime maps (in the key-value store) the path to the function call that generated the path. Whenever a path is required, the runtime consults the mapping to look up if that path was generated by a function call. If it was, then a function call dependency is inferred from the current executing function call to the function call that generated the path. For example, in listing 7.3, a call of `GenerateTable` generates the parse table file, which a call of `Parse` requires. The runtime then infers a dependency from the `Parse` call to the `GenerateTable` call. This is sound, because there may be at most one function call that generates a single path. We validate this property and abort execution when multiple function calls generate a single path. Therefore, we automatically infer hidden dependencies.

7.4.3 Reusing the Pluto Runtime

We have implemented our own API and runtime, instead of reusing the Pluto runtime, for the following three reasons. First of all, we reimplemented parts

of the Pluto runtime in order to better understand Pluto's incremental rebuild algorithm and concrete implementation. Second, we wanted to reduce boilerplate for writing foreign functions. Third, automated persistence would be hard to implement in Pluto, because Pluto requires every pipeline function to implement a `persistentPath` function (as seen in listing 7.1), which returns a unique filesystem path for persisting the result of executing a function with a particular input. We could generate a `persistentPath` implementation from the PIE DSL, but then foreign functions still need to manually implement this functions. Furthermore, filesystem paths may not contain certain characters, and have size limits (e.g., 260 characters on many Windows systems), which makes using files as a persistent storage complicated and error prone. Therefore, in the PIE runtime, we persist to a memory-mapped database.

7.5 PIE Language

In this section, we present PIE's language definition. We present PIE's syntax specification, describe domain-specific language constructs, and briefly look at static semantics. Finally, we describe compilation from the PIE language to the API, providing incremental and persistent pipeline execution when executed with the PIE runtime.

7.5.1 Syntax

Listing 7.4 shows PIE's syntax through an EBNF grammar specification. PIE programs are composed of (foreign) function definitions and foreign data types at the top level. Its constructs can be categorized into base constructs that can be directly translated to a general purpose language, and special constructs for the domain of interactive software development pipelines that require a special translation. Base constructs include regular unary and binary operations, control flow, list comprehensions, value declarations and references, function definitions and calls, early return or failure, literals, and string interpolation. Special constructs include path types, path literals, dependencies (`requires` and `generates`), and operations (`exists`, `read`, `list`, and `walk`); foreign function definitions and calls; and foreign data definitions.

We intentionally keep PIE's constructs simple in order to support incrementality and persistence, with concise expression of pipelines, while still supporting a wide range of different pipelines. For example, PIE does not allow assignment or other forms of mutation, because mutation complicates incrementality support. Instead, immutability allows the dynamic semantics to perform caching for improved incrementality.

7.5.2 Static Semantics

PIE is a statically typed and lexically scoped language. As base types, PIE has the unit type, booleans, integer, strings, paths, and user-defined foreign data types. Types can be made optional (`t?`), into a list (`t*`), and composed into tuples (`(t1, t2)`). All data type and function definitions are explicitly typed, but types are inferred inside function bodies. Static type checks prevent

```
idchr = ?[a-zA-Z0-9-_]?;
id    = {idchr};
qid   = {idchr | "."};
int   = ["-"]{?[0-9]?};

func_head = id "(" {id ":" t, ","} ")" "->" t;
func_def  = "func" func_head "=" ("foreign" id|"foreign java" qid "#" id|e);
data_def  = "data" id [":" id] "foreign java" id "{" {"func" func_head} "}";
program   = {func_def | data_def};

t = "unit"|"bool"|"int"|"string"|"path"| id | t "?" | t "*" | "(" {t, ","} ")";

e = "{" {e, ";"} "}" | "(" e ")"
  | "!" e | e "!" | e ("==" | "!=" | "||" | "&&" | "+") e
  | "if" "(" e ")" e ["else" e]
  | "[" e "|" binder "<-" e "]" | "val" binder "=" e
  | id | id "(" {e, ","} ")" | e "." id "(" {e, ","} ")"
  | "requires" e ["with" filter] ["by" stamper] | "generates" e ["by" stamper]
  | "exists" e | "read" e | "list" e ["with" filter] | "walk" e ["with" filter]
  | "return" e | "fail" e
  | "unit" | "true" | "false" | int | "null"
  | "(" {e, ","} ")" | "[" {e, ","} "]"
  | '"' {?~[\"\$\n\r]? | '\\$' | '\\"' | "$" id | "${" e "}"} '"'
  | ["."] "/" {?~[\n\r\$\,\;\]\)\ ]? | '\\ ' | '\\$' | "$" id | "${" e "}"};

binder = bind | "(" {bind, ","} ")";
bind   = id | id ":" t;

filter  = ("regex" | "pattern" ["s"] | "extension" ["s"]) e;

stamper = "exists" | "modified" | "hash";
```

Listing 7.4: PIE's syntax definition in a dialect of EBNF.

mistakes in the pipeline from appearing at runtime. For example, it is not possible to call a pipeline function with an argument of the wrong type, as PIE's type checker will correctly mark this as a type error. Name binding prevents mistakes such as duplicate definitions and unresolved references.

7.5.3 Compilation

To execute a PIE program with the PIE runtime, we compile it to a Kotlin program implementing the PIE API. We compile every function definition in the program to a class implementing `Func`, with corresponding input and output types, and compile its function to the `exec` method. Multiple function arguments, as well as tuple types, are translated into an immutable data class, implementing the required `equals`, and `hashCode` functions, and the `Serializable` interface. Function calls are compiled to `requireOutput` calls on the execution context, which records a function call dependency and incrementally executes that function.

Path dependencies are translated to `require` and `generate` calls on the execution context, which records path dependencies, and which infers hidden

dependencies when requiring a generated file. Path dependencies can use different *stampers*, which instruct the PIE runtime as to how generated and required paths are checked for changes during incremental execution. The `exists` stamper checks that a file or directory exists, `modified` compares the modification date of a file or directory, and `hash` compares the hash of a file, or the hashes of all files in a directory. The `exists`, `read`, `list`, and `walk` path operations are translated to function calls of built-in functions that perform these tasks and register the corresponding path dependencies. For example, the `walk` construct recursively walks over files and directories in a top-down fashion, returns them, and registers dependencies for each visited directory. Some path constructs also accept a *filter* that filters down the visited files and directories. For example, a `requires` on a directory with a filter only creates path dependencies for files and directories that are accepted by the filter. A regular expression, ANT pattern, or file extension filter can be used.

Other constructs (ones that do not affect incrementality or persistence) are compiled directly to Kotlin expressions. For example, list comprehensions are translated to `maps`.

7.6 Case Study: Spoofax Language Workbench

We evaluate PIE using two critical [43] case studies that are representative for the domain of interactive software development pipelines. In this section we discuss a case study in the domain of language workbenches. In the next section we discuss a case study in the domain of benchmarking.

Spoofax [75] is a language workbench for developing textual programming languages. Spoofax supports simultaneous development of a language definition and testing the programming environment generated from that language definition. This requires complex pipelines, including bootstrapping of languages [86]. In this case study we evaluate the feasibility of implementing the Spoofax pipeline using PIE.

In the Spoofax ecosystem, a programming language is specified in terms of multiple high-level declarative meta-language definitions, where each meta-language covers a language-independent aspect (e.g., separate syntax definition [153], name binding rules [5, 109, 89], or the dynamic semantics definition of a programming language [147]). Subsequently, Spoofax generates a complete implementation of a programming language, given all the meta-language definitions. Dividing a programming language implementation into linguistic abstractions in terms high-level meta-language definitions is the key enabler for maintainability of a language, however it complicates the necessary (interactive) software development pipelines.

Spoofax supports interactive language development in the Eclipse IDE, including developing multiple language specifications side-by-side. In contrast to a regular IDE that solely processes changes of source files in the source language, Spoofax additionally comes with support for interactive software development pipelines that respond to language specification changes. For example, changes to the syntax specification are reflected by reparsing source files of the language. In order to achieve this goal, Spoofax will: (1) execute

a pipeline to regenerate the language implementation based on the language specification, (2) reload the updated language implementation into the language registry, and (3) execute a pipeline for all open source files of the changed language.

The pipeline for source files will: (1) parse the source file into an AST and token stream, (2) generate syntax styling based on the token stream, (3) show parse errors (if any) and apply syntax styling, and (4) analyze and transform the source file.

7.6.1 Pipeline Re-Implementation

We have implemented Spoofax's management of multiple languages, parsing, and syntax-based styling with the PIE pipeline that is illustrated in listing 7.5. This is an extension to the example pipeline of section 7.3, but is still a subset of the complete pipeline due to space constraints. We omit the `foreign` keyword for brevity.

Language Specification Management. The first part of the pipeline is used to manage multiple language specifications. The `LangSpec` data type represents a language specification, which has a file extension and configuration required for syntax specification and styling. The `Workspace` type represents a workspace with multiple language specifications, which has a list of relevant file extensions, and a function to get the `LangSpec` for a `path` based on its extension. The aforementioned data types are similar to classes by binding function definitions to them. In this particular case their implementations are foreign (i.e., implemented in a JVM language), but registered in PIE in order for using them in an interactive software development pipeline. An instance of the `Workspace` (which contains `LangSpec`s) is created by the `getWorkspace` function from a configuration file. Interfacing with foreign functions and data types is a key enabler for embedding PIE pipelines in other programs, while still benefiting from domain-specific features such as dependency tracking.

Parse Table Generation, Parsing, and Styling. The second part implements parsing. There are several foreign data and function definitions which bind to Spoofax's tools. For example, `sdf2table` takes a specification in the SDF metalanguage, and produces a `ParseTable` which can be used to parse programs with the `jsglrParse` function. The `parse` function takes as input the text to parse and the language specification containing the syntax specification `mainFile` to derive a parser from, creates a parse table for the language specification, and uses that to parse the input `text`. Parsing returns a product type containing the `Ast`, `Token`s, and error `Message`s. Since parsing can fail, the AST and tokens are annotated with `?` multiplicity to indicate that they are nullable (optional). The third part implements syntax-based styling, similarly to parsing.

Processing Files in the IDE. The fourth part combines parsing and styling to process a single string or file and return the error messages and styling, which we can display in the Eclipse IDE. The fifth and sixth parts interface with the Eclipse IDE, by providing functions to keep an Eclipse project and editor up-to-date. A project is kept up-to-date by `walk`ing over the relevant files of

```
// 1) Language specification and workspace management
data LangSpec = {
  func syntax() -> path; func startSymbol() -> string; func styling() -> path
}
data Workspace = {
  func extensions() -> string*; func langSpec(path) -> LangSpec
}
func createWorkspace(string, path) -> Workspace
func getWorkspace(root: path) -> Workspace = {
  val text = read(root + "/workspace.cfg"); createWorkspace(text, root)
}
// 2) Creating parse tables and parsing
data ParseTable {} data Ast {} data Token {} data Msg {}
func sdf2table(path) -> ParseTable
func jsglrParse(string, string, ParseTable) -> (Ast?, Token*?, Msg*)
func parse(text: string, langSpec: LangSpec) -> (Ast?, Token*?, Msg*) = {
  val mainFile = langSpec.syntax(); requires mainFile;
  val startSymbol = langSpec.startSymbol();
  val table = sdf2table(mainFile); jsglrParse(text, startSymbol, table)
}
// 3) Syntax-based styling
data SyntaxStyler {} data Styling {}
func esv2styler(path) -> SyntaxStyler
func esvStyle(Token*, SyntaxStyler) -> Styling
func style(tokens: Token*, langSpec: LangSpec) -> Styling = {
  val mainFile = langSpec.styling(); requires mainFile;
  val styler = esv2styler(mainFile); esvStyle(tokens, styler)
}
// 4) Combine parsing and styling to process strings and files
func processString(text: string, langSpec: LangSpec) -> (Msg*, Styling?) = {
  val (ast, tokens, msgs) = parse(text, langSpec);
  val styling = if(tokens != null) style(tokens, langSpec) else null;
  (msgs, styling)
}
func processFile(file: path, langSpec: LangSpec) -> (Msg*, Styling?) =
  processString(read file, langSpec)
// 5) Keep files of an Eclipse project up-to-date
func updateProject(root: path, project: path) -> (path, Msg*, Styling?)* = {
  val workspace = getWorkspace(root);
  val relevantFiles = walk project with extensions workspace.extensions();
  [updateFile(file, workspace) | file <- relevantFiles]
}
func updateFile(file: path, workspace: Workspace) -> (path, Msg*, Styling?) = {
  val langSpec = workspace.langSpec(file);
  val (msgs, styling) = processFile(file, langSpec); (file, msgs, styling)
}
// 6) Keep an Eclipse editor up-to-date
func updateEditor(text: string, file: path, root: path) -> (Msg*, Styling?) = {
  val workspace = getWorkspace(root); val langSpec = workspace.langSpec(file);
  processString(text, langSpec)
}
```

Listing 7.5: Spoofax pipeline in PIE, with support for developing multiple language specifications, parsing, syntax styling, and embedding into the Eclipse IDE.

the project, and returning the messages and styling for each file which are displayed in Eclipse. An editor is kept up-to-date by processing the text in the editor.

7.6.2 Analysis

In this section we discuss the observations we made while re-implementing the incremental software development pipeline of Spoofax in PIE. Overall, the re-implementation improves on the areas mentioned below.

Canonical Pipeline Formalism. The main benefit over the old pipeline of Spoofax is that the PIE re-implementation is written in a single and concise formalism that is easier to understand and maintain. The old pipeline of Spoofax is comprised of code and configuration in four different formalisms: 1) Maven Project Object Model (POM) file that describes the compilation of Java source code, 2) an incremental build system using the Pluto [36] Java API and runtime that builds language specifications, 3) a custom (partially incremental) build system for building and bootstrapping meta-languages, and 4) a custom language registry that manages multiple language specifications. Incrementality and persistence are only partially supported, and implemented and maintained explicitly.

In contrast, the PIE pipeline is specified as a single formalism in a readable, concise, and precise way, without having to implement incrementality and persistence explicitly.

Exact (Dynamic) Dependencies. Spoofax's old pipeline emits dependencies that are either overapproximated or underapproximated, resulting in poor incrementality and therefore longer execution times. For example, in Spoofax, changing the styling specification will trigger parsing, analysis, compilation, and styling for all editors, even though only recomputation of the styling is required (i.e., sound overapproximation). On the other hand, changing the syntax specification will not trigger reparsing of files that are not open in editors (i.e., unsound underapproximation). In the PIE pipeline, these problems do not occur because of the implicit incrementality of function calls, and the right path dependencies.

For example, the `parse` function creates several dependencies which enable incremental recomputation. When the input `text`, `mainFile` path, contents of the `mainFile`, or the `startSymbol` changes, the function is recomputed. Furthermore, the function creates a parse table, which is a long-running operation. However, because of incremental recomputation and persistence, the parse table is computed once, and after that only when the syntax specification changes.

Support for Complex Pipeline Patterns. Due to space constraints, listing 7.5 omits the parts necessary for using Spoofax's name binding language and constraint solver, interfacing with existing Spoofax languages, and bootstrapping languages, but our re-implementation does support the aforementioned features. The full implementation can be found online [85].

```
func main(jmhArgs: string*) -> path* = {
  val jar = build(); val pkg = "io.usethesource.criterion";
  val javaSrcDir = ./src/main/java/io/usethesource/criterion;
  val benchs: (string, string, path*)* = [ // Benchmarks name, pattern, classes
    ("set","$pkg.JmhSetBenchmarks.(*)\$",[javaSrcDir+"/JmhSetBenchmarks.java"])
  , ("map","$pkg.JmhMapBenchmarks.(*)\$",[javaSrcDir+"/JmhMapBenchmarks.java"])
  ];
  val subjs: (string, string, path*)* = [ // Subjects name, identifier, libs
    ("clojure"     , "VF_CLOJURE"     , [./lib/clojure.jar     ])
  , ("champ"       , "VF_CHAMP"       , [./lib/champ.jar       ])
  , ("scala"       , "VF_SCALA"       , [./lib/scala.jar       ])
  , ("javaslang"   , "VF_JAVASLANG"   , [./lib/javaslang.jar   ])
  , ("unclejim"    , "VF_UNCLEJIM"    , [./lib/unclejim.jar    ])
  , ("dexx"        , "VF_DEXX"        , [./lib/dexx.jar        ])
  , ("pcollections", "VF_PCOLLECTIONS", [./lib/pcollections.jar])
  ];
  [run_benchmark(jar, jmhArgs, bench, subj) | bench <- benchs, subj <- subjs]
}
func build() -> path = {
  val pomFile = ./pom.xml; requires pomFile;
  [requires file | file <- walk ./src with extensions ["java", "scala"]];
  exec(["mvn", "verify", "-f", "$pomFile"]);
  val jar = ./target/benchmarks.jar;
  generates jar; jar
}
func run_benchmark(jar: path, jmhArgs: string*, bench: (string, string, path*),
  subj: (string, string, path*)) -> path = {
  val (bName, bId, bDeps) = bench; [requires dep | dep <- bDeps];
  val (sName, sId, sDeps) = subj;  [requires dep | dep <- sDeps];
  val csv = ./results/${bName}_${sName}.csv;
  requires jar by hash;
  exec(["java", "-jar", "$jar"] + bId + ["-p", "subject=$sId"] + jmhArgs +
    ["-rff", "$csv"]);
  generates csv; csv
}
```

Listing 7.6: Incremental performance benchmarking pipeline in PIE.

7.7 Case Study: Live Performance Testing

In this section we evaluate PIE on a case study for continuously monitoring the performance of a set of libraries. Specifically we use a snapshot of the *Criterion* benchmark suite [136] that measures the performance of immutable hash-set/map data structures on the JVM. The snapshot of Criterion was submitted as a well-documented artifact to accompany the findings of a research paper [137].

Under the hood, Criterion uses the Java Microbenchmarking Harness (JMH) [69] to execute benchmark suites against seven data structure libraries, producing Comma-Separated Values (CSV) files with statistical-relevant benchmarking data. Criterion uses bash scripts for orchestration, requiring to re-run all benchmarks whenever a benchmark or subject library changes. Those scripts are not able to exploit incrementality, which is tedious since benchmarking all combinations takes roughly two days, to produce statistically significant

outputs.

We re-engineered the pipeline such that initially each subject and benchmark combination is tested in isolation, and then incrementally re-execute all benchmarks for a particular subject if and only if that subject changes. In case the implementation of a benchmark changes, all subjects are re-tested for that benchmark. Regardless of the scenario, the CSV result files are kept up-to-date for subsequent data visualization.

We can apply such a pipeline on a local machine while developing the benchmarks for timely performance test results, or on a remote benchmarking server to minimize the amount of benchmarking work when something changes. While it is technically possible to write such an incremental pipeline in bash scripts, it would require a lot of manual work to implement, and will likely result in error-prone code. Fortunately, it is straightforward to write this pipeline in PIE.

7.7.1 *Pipeline Re-Implementation*

Listing 7.6 illustrates the benchmarking pipeline in PIE. The `build` function builds the benchmark and yields an executable JAR file `./target/benchmarks.jar`, by invoking Maven on the POM file `./pom.xml`. The `build` function `requires` all Java and Scala source files, to ensure that the JAR file is rebuilt as soon as a single source files changes.

To produce a CSV result file, the `run_benchmark` function executes the JAR file with the necessary command-line arguments for the JMH library, including the combination of `benchmark` and `subject`. The tuples `benchmark` and `subject` both store unique name identifiers —that are later used for naming the CSV file— and references to files they are comprised of. These file references are used by PIE to create dependencies for incremental re-execution.

Finally, `main` glues everything together by creating a list of benchmarks and subjects, running the benchmark with each combination of those, and by returning the up-to-date CSV files for subsequent data visualization.

7.7.2 *Analysis*

Compared to the existing bash script, the PIE pipeline provides incremental and persistent execution, and static analysis. The main benefit of the PIE pipeline over the bash script is that it provides incremental execution by function calls and path dependency annotations. In bash, implementing an incremental pipeline requires the pipeline developer to explicitly encode dependency tracking, change detection, caching, and more, which is why the existing bash script is not incremental. In the PIE pipeline, incrementality comes from stating the `requires` and `generates` dependencies in each function, which is straightforward because it is clear what the dependencies of each function are.

Furthermore, PIE performs static name and type analysis, before executing the pipeline, whereas bash has no static checks at all. This means that errors such as simple typographical errors, or appending a value of a wrong type to a list of strings, result in a static error in PIE which is easily fixed, but result in

	Make	Automake	OMake	Tup	PROM	Nix	Maven	Ant	Gradle	Jenkins	Shake	Pluto	Fabricate	Spark	Reactive Prog.	Workflow Lang.	PIE
Low Boilerplate	◒	◒	◒	◒	◒	◒	○	○	○	○	○	○	○	◒	○	●	●
Static Analysis	○	○	○	○	○	○	○	○	●	○	●	●	○	●	●	◒	●
Dynamic File Deps.	◒	●	●	◒	○	●	○	◒	○	○	◒	●	●	○	○	○	●
Implicit Incrementality	●	●	●	●	●	○	○	○	●	○	●	●	●	●	◒	○	●
Embeddable	○	○	○	○	○	○	○	○	○	○	●	●	●	●	●	○	●
Restartable	●	●	●	●	●	●	●	●	●	●	●	●	●	●	○	○	●
Cross-platform	◒	◒	●	●	○	●	●	●	●	●	●	●	○	●	●	●	●

Table 7.1: Feature overview of PIE and related work (● = full support, ◒ = partial/limited support, ○ = no support).

run-time errors in bash.

7.8 Related Work

In this section we discuss related work with a focus on build systems. Table 7.1 provides a feature overview of the systems we discuss throughout this section.

7.8.1 Partial Domain-Specific Build Abstractions

Make [133] is a build automation tool based on declarative rules. Make extracts a static dependency graph from these rules, and executes the commands according to the dependency graph. Upon re-execution, Make is able to detect unchanged files that do not require regeneration based on timestamps. Make supports a limited form of dynamic dependencies that does not generalize, i.e., an include directive that allows loading other Makefiles.

Automake [98] alleviates many of Make's shortcoming by introducing a formalism on top of Make that generates Makefiles. Automake is mostly geared towards C compilation and other compilation processes that follow similar patterns, but cannot be used to write arbitrary interactive pipelines, making it less flexible than PIE. Due to a lack of static checking, ill-typed Automake scripts may propagate defects — that are only detectable at run time — to the generated Makefiles. In contrast, PIE catches such errors statically before pipeline execution.

OMake [62] is a build tool with a Make-like syntax, but with a richer dependency tracking mechanism. PIE is similar to OMake in that both supports a form of dynamic path dependencies (called side-effects in OMake), and incrementality based on these dependencies. However, like Make, OMake works exclusively with files and command-line processes, meaning that it is not possible to depend on the result of a function call, or to interface with foreign functions and types, making it unsuitable for interactive pipelines

which require embedding into an interactive system.

Tup [127] is a build tool with Make-like rules. Tup automatically infers required file dependencies by instrumenting the build process, providing more fine-grained dependencies than Make. However, the dependency on the input file and generated file must still be declared statically upfront.

PROM [79] is a Prolog-based make tool where Make-like builds are specified declaratively and executed as Prolog terms, increasing expressiveness and ease of development. PROM's update algorithm executes in two phases, where first a file-dependency graph is created, after which creation rules are executed to create new files, or to update out-of-date files. Because of these phases, PROM does not support dynamic discovery of dependencies during build execution.

Nix [28, 30] is a purely functional language for building and deploying software. One of its applications is managing the configuration of the operating system NixOS [29]. Nix supports incremental execution of pipelines through cryptographic hashes of attributes and files, but must be explicitly initiated by the developer through the use of the `mkDerivation` function. While incrementality becomes explicit, Nix puts the burden on pipeline developers, whereas PIE supports incrementality implicitly. Furthermore, Nix is dynamically checked, meaning that name and type defects are reported at runtime, as opposed to before runtime with static checking in PIE.

Maven [45] is a software dependency management and build tool, popular in the Java ecosystem. It features a fixed sequence of pipeline steps such as compile, package, and deploy, which are configured through an XML file. Maven is neither incremental nor interactive, requiring a full batch re-execution every time data in the pipeline changes.

Ant [44] is a build automation tool, using XML configuration files for defining software development pipelines. Ant supports incrementality by inserting `uptodate` statements that check if a source file is up to date with its target file, making incrementality explicit, at the cost of burdening the developer. Ant does not provide static analysis.

Gradle [67] is a build automation tool, programmable with the Groovy language, featuring domain-specific library functions to specify builds declaratively. Gradle supports incremental task execution through annotations that specify a task's input/output variables, files, and directories. Like Make, dependencies have to be specified statically up-front, causing an overapproximation of dependencies.

Jenkins is a continuous integration server which can be programmed with its Groovy pipeline and a set of domain-specific library functions [70]. Jenkins can detect changes to a (remote) source code repository to trigger re-execution of an entire build pipeline, however without support for incrementality.

7.8.2 Software Development Pipelines as a Library

Subsequently discussed software pipeline solutions are available as a library (i.e., internal DSL) implemented in a general-purpose programming language. Unlike an external DSL solution such as PIE, those libraries do not support domain-specific syntax or error reporting in terms of the domain, instead

requiring encoding of domain concepts. Furthermore, it is hard, if not impossible, to restrict features of a programming language via a library, that heavily influence incrementality, such as mutable state. Finally, the compiler of the PIE DSL can be retargeted to a different environment or programming language, enabling a PIE pipeline to be embedded into different interactive environments without (or minimal) alteration.

Shake [107, 108] is a Haskell library for specifying build systems. Unlike Make, required file dependencies can be specified during builds in Shake, supporting more complex dependencies and reducing overapproximation of dependencies. However, like Make, targets (generated file dependencies) have to be specified up-front. This means that it is not possible to specify builds where the names of generated files are decided dynamically. For example, the Java compiler generates a class file for each inner class in a source file, where its file names are based on the inner and outer class name. Therefore, the generated file dependencies of the Java compiler are decided dynamically, and cannot be specified in Shake.

Pluto [36] is a Java library for developing incremental builds, which we have already discussed extensively in section 7.2. One difference between Pluto and PIE is that Pluto supports incremental cyclic builders, whereas PIE does not. We have opted not to implicitly support cycles for simplicity of the build algorithm, and because cycles typically do not appear in pipelines. Cycles can be handled explicitly in PIE by programming the cyclic computation inside a single pipeline function.

Fabricate [64] is a Python library for developing incremental and parallel builds, that aims to automatically infer all file dependencies by tracing system calls. System call tracing is not cross-platform, only fully supporting Linux at the moment. PIE in contrast is cross-platform, because its runtime works on any operating system the JVM runs on.

Apache Spark [7] is a big data processing framework where distributed datasets are transformed by higher-order functions. PIE is similar to Spark in that both create dependency graphs between calculations. For example, when transforming a dataset with Spark (e.g., with map or filter), the derived dataset depends on the parent dataset, such that the derived dataset is rederived when the parent changes. PIE differs from Spark in that PIE works with local data only, whereas Spark works with a distributed storage system required for big data processing. However, PIE supports arbitrary computations (as opposed to a fixed set of higher-order functions in Spark), and dynamic file dependencies.

7.8.3 General-Purpose Languages

Reusing an existing general-purpose language, such as Java or Haskell, and giving it an incremental and persistent interpretation is not feasible for several reasons. It requires adding additional constructs to the language, such as path dependencies and operations, which require changes to the syntax, static semantics, dynamic semantics (compiler or interpreter) of the language. That requires at the very least being able to change the language, which is not always possible. Even when it is possible, language parsers, checkers, and compilers

are often large codebases that require significant effort to change. Furthermore, we also need to ensure that existing constructs work under incrementality. For example, mutable state in Java interferes with incrementality.

7.8.4 Reactive Programming

Reactive programming is characterized by asynchronous data stream processing, where data streams form a pipeline by composing streams with a set of stream combinators. Reactive programming approaches come in the form of libraries implemented in general-purpose languages, such as Reactive Extensions [101], or as an extended language such as REScala [126]. Reactive programming approaches provide a form of incrementality where the reactive pipeline will rerun if any input signal changes. However, they do not cache outputs or prevent re-execution of pipeline steps when there are no changes. Note that reactive programming approaches operate in volatile memory only, whereas PIE's runtime supports persistence (i.e., pause and resume) of pipeline executions. Preserving a pipeline's state is of special importance in interactive environments such as IDEs, to support restarting the programming environment without re-triggering potentially expensive calculations. Furthermore, reactive programming approaches do not support (dynamic) file dependencies.

7.8.5 Workflow Languages

Workflows, like pipelines, describe components (processors) and how data flows between these components. Workflow languages are DSLs which are used to model data analysis workflows [4], business process management [1], and model-to-model transformations [12], among others. The crucial differentiation between many workflow systems and software development pipelines, is that the former model manual steps that require human interaction, whereas the latter focuses on processors that perform general purpose computations.

7.9 Future Work

We now discuss directions for future work.

7.9.1 First-Class Functions and Closures

Currently, the PIE DSL does not support first-class functions and closures, for simplicity. The PIE runtime does support first-class functions, since function calls are immutable and serializable values which can be passed between functions and called. However, closures are not yet supported, because (again for simplicity), functions must be registered with the runtime before a pipeline is executed.

To fully support first-class functions and closures, we must add them to the PIE DSL, and support closures in the runtime and API. This requires closures to be serializable, which the JVM supports. Closures from foreign functions must ensure not to capture mutable state, non-serializable values, or large objects graphs, as these can break incrementality. Spores [105] could be used to guarantee these properties for closures.

7.9.2 *Live Pipelines*

PIE pipelines can dynamically evolve through the inputs into the pipeline: files on the filesystem such as configuration files, and objects passed through function calls such as editor text. However, the pipeline code itself currently cannot dynamically evolve at runtime. When the pipeline code itself is changed, the pipeline must be recompiled and reloaded. This process is relatively fast, because compiling PIE DSL code and restarting the JVM is fast, but can be improved nevertheless. Furthermore, dynamic evolution of pipelines at runtime is especially important if we want to apply PIE to live programming environments.

While there are known solutions for compiling and reloading code in the JVM, such as using class loaders, it is unclear how to handle incrementality in the face of changes to the pipeline program. For example, if the `normalize` function in listing 7.2 is changed, all calls of `normalize` are potentially out-of-date and need to be re-executed, as well as all function calls that (transitively) call `normalize`. Similarly, foreign functions and data types can be changed, which require re-execution or even data migrations in the persistent storage.

7.10 Conclusion

We have presented PIE: a DSL, API, and runtime for developing interactive software development pipelines. PIE provides a straightforward programming model that enables direct and concise expression of pipelines with minimal boilerplate, reducing the development and maintenance effort of pipelines. Compared to the state of the art, PIE reduces the code required to express an interactive pipeline by a factor of 6 in a case study on syntax-aware editors. Furthermore, we have evaluated PIE on two complex interactive software development pipelines, showing that the domain-specific integration of features in PIE enable concise expression of pipelines, which are normally cumbersome to express with a combination of traditional build systems and general-purpose languages.

Acknowledgements

This research was supported by NWO/EW Free Competition Project 612.001.114 (Deep Integration of Domain-Specific Languages) and NWO VICI Project (639.023.206) (Language Designer's Workbench).

Scalable Incremental Building with Dynamic Task Dependencies

8

Abstract

Incremental build systems are essential for fast, reproducible software builds. Incremental build systems enable short feedback cycles when they capture dependencies precisely and selectively execute build tasks efficiently. A much overlooked feature of build systems is the expressiveness of the scripting language, which directly influences the maintainability of build scripts. In this chapter, we present a new incremental build algorithm that allows build engineers to use a full-fledged programming language with explicit task invocation, value and file inspection facilities, and conditional and iterative language constructs. In contrast to prior work on incrementality for such programmable builds, our algorithm scales with the number of tasks affected by a change and is independent of the size of the software project being built. Specifically, our algorithm accepts a set of changed files, transitively detects and re-executes affected build tasks, but also accounts for new task dependencies discovered during building. We have evaluated the performance of our algorithm in a real-world case study and confirm its scalability.

8.1 Introduction

Virtually every large software project employs a build system to resolve dependencies, compile source code, and package binaries. One great feature of build systems besides build automation is incrementality: After a change to a source or configuration file, only part of a build script needs re-execution while other parts can be reused from a previous run. Indeed, incremental build systems are a key enabler for short feedback cycles. The reliable and long-term maintainable usage of incremental build systems requires the following three properties:

Efficiency. The most obvious requirement is that rebuilds must be efficient. That is, the amount of time required for a rebuild must be proportional to how many build tasks are affected by a change. Specifically, a small change affecting few tasks should only incur a short rebuild time.

Precision. An incremental rebuild is only useful if it yields the exact same result as a clean build. To this end, incremental build systems must capture precise dependency information about file usage and task invocations. Make-like build systems do not offer means for capturing precise dependencies. Instead, over-approximation (`*.h`) leads to inefficiency because of considering too many files, and under-approximation (`mylib.h`) leads to incorrect rebuilds because of missing dependencies (e.g., `other.h`). Precise dependency information is

required for efficient and correct rebuilds.

Expressiveness. Like all software artifacts, build scripts grow during a project's lifetime [100] and require increasing maintenance [93]. Therefore, build scripts should be written in expressive languages, avoiding accidental complexity. That is, build scripting languages should not require build engineers to apply complicated design patterns (e.g., recursive [106] or generated Makefiles) for expressing common scenarios.

Current incremental build systems put a clear focus on efficiency and precision, but fall short in terms of expressiveness. In particular, in order to support incremental rebuilds, current systems impose a strict separation of configuration and build stages. All variability of the build process needs to be fixed in the configuration stage, whereas the build stage merely executes a pre-configured build plan. This model contradicts reality, where *how to build an artifact* depends on the execution of other build tasks. We have observed two sources of variability in building. First, based on the result of other tasks, *conditional building* selects one of multiple build tasks to process a certain input. Second, based on the result of other tasks, *iterative building* invokes build tasks multiple times on different inputs. In both cases, dependencies on task invocations only emerge during the build; build engineers cannot describe these *dynamic dependencies* in the configuration phase. We illustrate a concrete example in section 8.2.

A solution to the expressiveness problem is to provide build engineers with a full-fledged programming language. In such a system, build tasks are procedures that can invoke other build tasks in their body. Build tasks can inspect the output of invoked tasks and use that to conditionally and iteratively invoke further tasks. The problem of such a programmable build system is that it is difficult to achieve incrementality. We are only aware of a single build system that is both programmable and incremental: Pluto [36]. Unfortunately, the incremental build algorithm of Pluto has an important limitation: To check which tasks need re-execution, Pluto needs to traverse the entire dependency graph of the previous build and has to touch every file that was read or written in the previous build. This contradicts our first requirement, *efficiency*, because the rebuild time of Pluto depends on the size of the software project more than it depends on the size of the change. In particular, even when no file was changed, Pluto's algorithm requires seconds to determine that indeed no task requires re-execution. We illustrate Pluto's algorithm using an example in section 8.2.

In this paper, we design, implement, and evaluate a new incremental build algorithm for build systems with dynamic task dependencies. While Pluto's algorithm only takes the old dependency graph as input and traverses it top-down, our algorithm also takes a set of changed files and primarily traverses the dependency graph bottom-up. We can collect changed files, for example, from IDEs that manage their workspace or by using a file system watchdog. Our algorithm uses the changed files to drive rebuilding of tasks, only loading and executing those tasks that are (transitively) affected by a change. However,

due to dynamic task dependencies, the dependency graph can change from one build to the next one. Our build algorithm accounts for newly discovered and deleted task dependencies by mixing bottom-up and top-down traversals.

Our new incremental build algorithm provides significant performance improvements when changes are small. We have conducted a real-world case study on the Spoofax language workbench, a tool built for developing domain-specific languages (DSLs). The build script of Spoofax processes DSL specification files and generates interpreters, compilers, and IDE plug-ins for them. We found that our algorithm successfully eliminates the overhead of large dependency graphs and provides efficient rebuilding that is proportional to the change size.

In summary, we make the following contributions. We review programmatic build scripts, incremental building with Pluto, and why this does not scale (section 8.2). We describe our key idea of bottom-up incremental building, and what is needed to make it work (section 8.3) We present our hybrid incremental build algorithm that mixes bottom-up and top-down building (section 8.4), and briefly discuss its implementation (section 8.5). We evaluate the performance of the hybrid algorithm against Pluto's algorithm with a case study on the Spoofax language workbench (section 8.6).

8.2 Background and Problem Statement

Most build systems provide a *declarative* scripting language. Declarative languages are great as they let developers focus on *what* to compute rather than *how* to compute it. However, we argue that declarativity is misdirected when it comes to describing sophisticated build processes that involve conditional and iterative task application.

For example, consider the build script in listing 8.1. We wrote this build script in the PIE build script language [91], which mostly provides standard programming language concepts. That is, the build script performs iterative building by defining and calling functions (tasks) like `main` and `parseYaml`, stores results of tasks in local variables such as `config` and `src` which can be immediately used by subsequent tasks, and involves conditional building with control structures like `if` and `for`. By and large, our build script is a normal program that happens to handle file paths and invoke external processes to generate and run tests. But how can we execute such a programmatic build script incrementally?

Most build systems require declarative specifications of build tasks for this reason: to support efficient incremental rebuilds. However, Erdweg et al. demonstrated that it is also possible to incrementally execute programmatic build scripts, with *Pluto* [36], a build system that incrementally executes build scripts written in Java. The PIE language we used in our example is an alternative front-end to Pluto [91].

The build algorithm of Pluto constructs a dependency graph of a build while the build script runs. For example, consider the dependency graph of our example script in fig. 8.1. The dependency graph contains a node for each invoked task and for each read or written file. Edges between nodes encode

```
func main() -> string {
  val config = parseYaml(./config.yaml)
  val src = config.srcDir
  if (config.checkStyle) {
    val styleOk = checkStyle(src)
    if (config.failOnStyle && !styleOk)
      return "style error"
  }
  val userTests = ./test/**
  val genTests = genTests(src, ./test-gen)
  var failed = 0
  for (test <- userTests ++ genTests) {
    val testOk = runTest(test)
    if (!testOk) failed += 1
  }
  return "Failed tests: " + failed
}
func parseYaml(p: path) -> Config {...}
func checkStyle(src: path) -> bool {...}
func genTests(src: path, trg: path) -> path* {...}
func runTest(test: path) -> bool {...}
```

Listing 8.1: Build script that invokes tasks conditionally and iteratively at build time.

dependencies. A task depends on the tasks it invokes and on the files it reads or writes. Moreover, when a task reads a file that was generated by another task, the reading task depends on the generating task such that the generating task is executed first. While not shown in our graph, both task-task edges and task-file edges are labeled with stamps (e.g., timestamp, hashsum) that determine if the task output respectively file content is up-to-date.

The incremental build algorithm of Pluto takes the dependency graph of the previous run and selectively reruns tasks to ensure consistency of the build. The dependency graph of a build is consistent if for each invoked task (i) all read and written files are up-to-date and (ii) the outputs of all called tasks are up-to-date. An incremental build algorithm is correct if it always restores consistency [36]. Or intuitively: a correct incremental build algorithm yields the same result as a clean build. The challenge is to restore consistency with as little computational effort as possible. For example, let us assume the initial dependency graph (top-left in fig. 8.1) is consistent to begin with. We discuss three different changes **C1**–**C3**:

C1. If we change file `./config.yaml` to turn off style checking, task `main` becomes inconsistent since the stamp of its file dependency changes (e.g., newer timestamp, changed hashsum). We can restore consistency by rerunning tasks `parseYaml` and `main` only; all other tasks remain consistent since they neither invoke `main` nor read files written by `main`. The incremental build yields a new dependency graph (top-right in fig. 8.1), where the new invocation of `main` does not depend on `checkStyle` anymore.

C2. If instead, we change the content of user test `./test/X`, task `runTest(./test/X)` becomes inconsistent and needs rerunning. Since task `main`

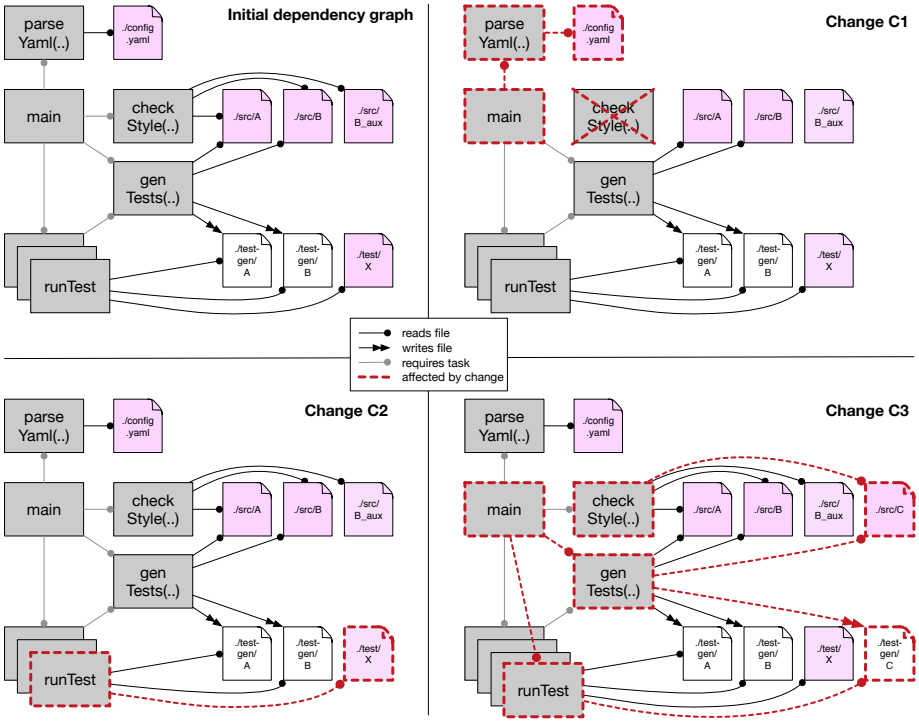


Figure 8.1: The dependency graph of a build captures task and file dependencies and is the basis for incremental building.

calls `runTest(./test/X)`, it needs rerunning if the boolean value returned by `runTest` flips. Either way, the dependency graph remains unaffected (bottom-left in fig. 8.1).

C3. Finally, if we add a source file `./src/C`, tasks `checkStyle` and `genTests` are affected because they depend on the directory `./src`. If the new file has a style error, this affects `main` and yields a new dependency graph where no testing occurs (not shown). Otherwise, let us assume `genTests` produces a new test `./test-gen/C` for the added file, which affects `main`'s scan of directory `test-gen`. During the subsequent rerun of `main`, we discover a new task invocation `runTest(./test-gen/C)` (bottom-right in fig. 8.1).

The incremental build algorithm of Pluto can handle these and any other changes correctly. Moreover, the algorithm is optimally incremental in the sense that it only executes a task if absolutely necessary. The basic idea of the algorithm is to start at the root node(s) of the dependency graph, to traverse it depth-first, and to interleave consistency checking and rerunning. In particular, while rerunning a task that invokes another task, if the invoked task exists in the dependency graph, continue with consistency checking of the invoked task and only rerun it if necessary. This interleaving of consistency

checking and rerunning is what enables support for conditional and iterative task invocations.

Problem Statement. The incremental build algorithm of Pluto has an important limitation. Irrespective of the changed files, it has to traverse the entire dependency graph to check consistency and to discover tasks that need rerunning. While that may be fine for larger changes that affect large parts of the graph like **C3**, the overhead for small-impact changes like **C2** is significant. Especially in interactive settings, where developers routinely trigger sequences of small-impact changes, this overhead can quickly render the system unresponsive. The problem is that the algorithm does not *scale down* to small changes when *scaling up* to large dependency graphs.

Our goal is to design, realize, and evaluate a new incremental build algorithm for programmatic build scripts that scales in the size of a change. That is, rebuild times should be proportional to the impact a change has on the overall build. In particular, rebuild times should be independent of the size of the dependency graph. These requirements preclude a full traversal of the dependency graph to discover affected tasks as done by Pluto. Instead, our new algorithm takes the set of changed files as input and only ever visits affected tasks.

8.3 Key Idea and Challenges

The key idea to increasing the scalability of the Pluto algorithm is to execute tasks *bottom-up*. In this section, we motivate this approach, and we discuss the corner cases that require adjustments to a pure bottom-up algorithm.

8.3.1 Bottom-Up Traversal

The key problem of the Pluto build algorithm is that it visits and checks tasks that are ultimately unaffected. For example, in change **C2** in section 8.2, only a single task (`runTest`) is affected by the change to file `./test/X`. However, Pluto will visit and check *all* reachable tasks in a top-down depth-first traversal, including the tasks that are not affected by the change (`parseYaml`, `checkStyle`, and `genTests`). Establishing that these tasks *are unaffected* is expensive, as our benchmarks demonstrate (section 8.6).

To make the algorithm scale, it should only visit the nodes of the dependency graph that are actually affected by a change. The changes that trigger a re-build are to *files*, which are at the *leaves* of the dependency graph. Tasks that need to be recomputed depend directly or indirectly on such file changes. Instead of looking for tasks that may indirectly depend on a change and gradually getting closer to the actual change, as Pluto does, why not start with those changes and the tasks that depend on them?

The key idea of our algorithm is to *traverse the dependency graph bottom-up*, driven by file changes, only visiting and checking affected tasks. The algorithm first executes the tasks that are *directly* affected by changed files. For example, in change **C2**, file `./test/X` changes, which directly affects task `runTest(./test/X)`, which must therefore be re-executed. Tasks can also be

indirectly affected by a file change, namely when it reads a file produced by an affected task or when it reads the output value of an affected task. For example, in change **C3**, file `./src/C` is added, which triggers re-execution of `genTests`, which yields a new output value to main, which thus is indirectly affected, re-executes, and creates a new `runTest` task. A subsequent edit of file `./src/C` triggers `genTests` again, which produces the same value as before but updates the generated file `./test-gen/C`, which affects the corresponding `runTest` task (the `main` task is not affected this time).

Thus, a bottom-up traversal executes tasks that are affected by changed files or by other affected tasks, following a path from the changed leaves of the dependency graph to the root(s). However, a pure bottom-up traversal is not adequate to support programmatic build scripts with dynamic dependencies. We discuss the adjustments that are necessary to realize an adequate algorithm.

8.3.2 Top-Down Initialization

In order to perform a bottom-up traversal over the dependency graph, we need a dependency graph to start with. Therefore, we start with Pluto's top-down algorithm to obtain the initial dependency graph. This is efficient, since every task is affected in the initial build.

8.3.3 Early Cut-Off

By default, a bottom-up traversal takes the transitive closure of dependencies, re-executing all tasks on the path from a changed file to the root(s) of the dependency graph. However, re-execution of a task does not always lead to a new result. If the result was the same as before, the path to the root can be cut off early. For example, in **C2** `main` depends on `runTest(./test/X)`, which depends on the changed `./test/X` file. So, do we need to re-execute `main`? That depends on the output of task `runTest(./test/X)`. If the result is a different (integer) value than before, the number of tests that fail changes, and `main` should be re-executed Otherwise, `main` is not affected and we can cut off the build early[1], as shown in the bottom-left part of fig. 8.1.

8.3.4 Order of Recomputation

Another potential problem of naive bottom-up evaluation is that tasks may be executed multiple times. For example, in change **C3**, `main` depends on two existing affected tasks: `checkStyle` and `genTests`. A possible execution trace when `checkStyle` *does affect* `main` (not shown in the figure), is to execute `checkStyle`, then `main` which is affected by `checkStyle`, then execute `genTests`, and then execute `main` again because it is affected by `genTests`. Executing a task multiple times is not only inefficient, but also causes glitches: inconsistent results that are exposed to users.

To avoid such re-executions, we should ensure that all affected dependencies of a task are executed before the task itself. Instead of eagerly executing tasks

[1]In a real-world build script, `runTest` would output a report of which tests fail and why, and `main` would be re-executed whenever this changes. We support this, but chose to keep the example from section 8.2 simple for demonstration purposes.

when encountered during a bottom-up traversal, we *schedule* tasks in a priority queue, which is topologically sorted according to the dependency graph. Until the queue is empty, scheduled tasks are removed from the front of the queue and executed. The topological ordering of the priority queue ensures that task dependencies are executed before the task itself.

8.3.5 Dynamic Dependencies

The final challenge is to support dynamic dependencies during a bottom-up traversal. Consider change **C3** again, where a new task `runTests(./test-gen/C)` is discovered by `main`. A bottom-up traversal can never detect such a dynamic dependency, since it only has access to the dependency graph of the previous run. To remedy this, we temporarily switch to top-down depth-first building when executing a task, so that the task can discover dependencies to new tasks, discover dependencies to existing tasks, or remove existing dependencies.

When discovering a dependency to a new task, top-down depth-first building continues recursively by (eagerly) executing the new task. For example, in change **C3**, task `main` is (indirectly) affected and thus is built in a top-down manner (after its dependencies `checkStyle` and `genTests` have been built), which recursively calls task `runTests(./test-gen/C)` and registers a dependency to it. Furthermore, when a dependency is discovered to a task t that exists in the old dependency graph, it might be affected already. That is, t and dependencies of t may have been scheduled in the queue. We cannot execute t before executing its dependencies. Therefore, we temporarily switch back to bottom-up building, executing dependencies of t that are scheduled in the queue, until t itself is executed or found unaffected. Then we switch back to top-down building and continue executing the caller. Finally, when a dependency is removed (i.e., dependency to a task that was made in the previous run, but not in this run), the dependency graph is updated, but no further action is taken.

8.3.6 Dependency Graph Validation

As in the Pluto algorithm, we also need to enforce validity of the dependency graph by detecting overlapping generated files, hidden task dependencies, and cyclic tasks. An overlapping generated file occurs when more than one task generates (creates or writes to) the same file. This makes it unclear in which order those tasks must be executed to bring the file into a consistent state, and is therefore disallowed. Furthermore, a hidden dependency occurs when a task requires (reads) a file that was generated by another task, without the requiring task depending on the generator task. Such a dependency must be made explicit, so that the generated file is updated by the generator task before being read by the requiring task. Finally, a task is cyclic when it (indirectly) calls itself. We disallow cyclic tasks to ensure termination of the build algorithm. We check these invariants on-the-fly while constructing the new dependency graph for subsequent incremental builds.

```
var T_q
var T_e
var O_c
var DG_new
function buildNewTask(t, DG_old)
  T_e := ∅
  O_c := ∅
  DG_new := DG_old
  exec(t)
function buildWithChangedFiles(F, DG_old)
  T_e := ∅
  O_c := ∅
  DG_new := DG_old
  T_q := PriorityQueue(DG_old.depOrder())
  schedAffByFiles(F, DG_old)
  while T_q ≠ ∅ do
    execAndSchedule(T_q.poll(), DG_old)
```

(a) Variables and main build functions.

```
function execAndSchedule(t, DG_old)
  val r := exec(t)
  schedAffByFiles(r.genFiles, DG_old)
  schedAffCallersOf(t, r.output, DG_old)
  return r.output
function schedAffByFiles(F, DG_old)
  for f ← F do
    for (stamp, t) ← DG_old.requireesOf(f) do
      if ¬stamp.isConsistent(f) then
        T_q := T_q ∪ t
    if (stamp, t) ← DG_old.generatorOf(f) then
      if ¬stamp.isConsistent(f) then
        T_q := T_q ∪ t
function schedAffCallersOf(t, o, DG_old)
  for (stamp, t_call) ← DG_old.callersOf(t) do
    if ¬stamp.isConsistent(o) then
      T_q := T_q ∪ t_call
```

(b) Change-driven, bottom-up building.

```
function exec(t)
  if t ∈ T_e then abort
  T_e := T_e ∪ t; val r := t.run(); T_e := T_e \ t
  DG_new := DG_new ∪ r; validate(t, r); observe(t, r.output)
  O_c[t] := r.output; return r.output
function require(t, DG_old)
  if o ← O_c[t] then return o
  else if t ∈ DG_old then return requireNow(t, DG_old)
  else return exec(t)
function requireNow(t, DG_old)
  while val t_min := T_q.leastDepFromOrEq(t) do
    T_q := T_q \ t_min
    val o := execAndSchedule(t_min, DG_old)
    if t = t_min then return o
  val o := DG_old.outputOf(t)
  observe(t, o); O_c[t] := o; return o
function validate(t, r)
  for f ← r.genFiles do
    for (_, t_gen) ← DG_new.generatorOf(f) do
      if t ≠ t_gen then abort
  for f ← r.reqFiles do
    for (_, t_gen) ← DG_new.generatorOf(f) do
      if ¬DG_new.callsTaskTr(t, t_gen) then abort
```

(c) Execution, requirement, and validation.

Listing 8.2: Change-driven incremental build algorithm.

8.4 Change-Driven Incremental Building

In this section, we present our *hybrid algorithm* that mixes bottom-up and top-down incremental building based on the observations and ideas from the previous section. We present the algorithm in three parts: main functions (listing 8.2a), bottom-up building (listing 8.2b), and execution (listing 8.2c). All functions share four global variables defined at the top of listing 8.2a. Variable T_q is a topologically ordered priority queue of affected tasks that still need to be executed. Variable T_e is a set of currently executing tasks, used to detect cyclic tasks. Variable O_c is a cache of output values for tasks that have already been executed. Finally, variable DG_{new} is the new dependency graph that is constructed from the old dependency graph and dynamic dependencies on-the-fly.

We provide two entry points to incremental building in listing 8.2a, both of which first clear the set of executing tasks T_e, clear the cache O_c, and copy the old dependency graph DG_{old} to DG_{new}. Function buildNewTask is the entry point for an initial build. Function buildNewTask then simply invokes function exec (listing 8.2c) to execute the task. We describe exec below.

The second entry point buildWithChangedFiles is more interesting as it initiates bottom-up building. It takes as input a set of changed file paths F, represented as filesystem path strings such as `./config.yaml`, and the old dependency graph DG_{old}. The basic idea is to schedule and run affected tasks using priority queue T_q until all affected tasks are up-to-date. To this end, we create a new priority queue using the task dependencies in DG_{old} as a topological ordering. We call function schedAffByFiles (described below) with the old dependency graph to find all tasks *directly* affected by the changed file paths F, and add those tasks to the queue T_q. The main loop of bottom-up building is the following *while*-loop: As long as there are affected tasks in the queue, poll a scheduled task (retrieve the task at the front and remove it) from the queue, execute it, and add all tasks affected by it to the queue. Since the queue is topologically ordered, dependencies of tasks are executed before the task itself. Unless a task itself does not terminate (for example by recursively calling new tasks ad infinitum), the queue becomes empty at some point since cyclic tasks are disallowed, terminating the algorithm.

8.4.1 Bottom-Up Building

Whenever a task occurs in the priority queue T_q, it is definitely affected (directly or indirectly) by changed files. Hence, no further consistency check is necessary. Function execAndSchedule in listing 8.2b accepts an affected task, runs it unconditionally using exec, and schedules new tasks based on the generated files and output value of the executed task. If a task does not change or create new generated files, nor produce a new output value, no new tasks will be scheduled and building may be cut off early.

Function schedAffByFiles schedules tasks based on changed file paths F. If task t requires a changed file and the stamp *stamp* has changed (is inconsistent) then t is affected by the change to f and is scheduled by adding it to T_q.

Analogously, if a task generates a file that has changed, it is affected and thus scheduled. A stamp contains a summary of a file's content, such as the last modification date or a hash, and is used to efficiently check whether a file has changed with isConsistent. For example, when using the file's modification date as a stamp, we compare the modification date in the stamp, with the current modification date of the file on the local filesystem, and consider the file changed if the modification date is different. We use the old dependency graph DG_{old} (computed in a previous run of the algorithm) to find tasks that require a file (requireesOf), and to find the task that generates a file (generatorOf), along with the stamp that was produced at the time the dependency was created.

Likewise, the schedAffCallersOf function schedules callers of task t based on changes to its output value o. If t_{call} has a dependency to task t, and that dependency is inconsistent with relation to the new output value o of t, then t_{call} is affected by the new output value o and is scheduled. Similarly, we use a *stamp* of the output value, which could be the full output value, such as an integer representing the number of failing tests, or a summary of the value such as a hash, and compare the stamp with the new output value with isConsistent. Finally, the old dependency graph DG_{old} is used to find callers of a task with callersOf.

8.4.2 Execution, Requirement, and Validation

Function exec (listing 8.2c) executes the body of t. During task execution, a task may require (call) other tasks with the require function. Therefore, we first need to check if we are already executing task t, and abort when a cycle is detected. Then, we add t to the set of executing tasks T_e, run the body of the task, and remove t from T_e. Once execution completes, we update the new dependency graph DG_{new} with the result r of executing t. A result r contains the dynamic dependencies the task made during execution: a set *reqFiles* of read files, *genFiles* of created or written to files, and a set *reqTasks* of other tasks that were called by t; and the output value *output* that the task produced. A dependency graph DG is a set of those results, where each task has a single result. We then validate the new dependency graph, call any external observers of the task's output with observe, cache the output, and finally return the output.

We use function exec to execute tasks both during bottom-up and top-down traversals. While exec is agnostic to the traversal order, function require must take care to handle tasks required bottom-up and top-down correctly. We distinguish three cases. If t was already executed (visited) this run, we return its cached output value $O_c[t]$. Otherwise, we check if t was in the old dependency graph DG_{old}. If task t is new and does *not* occur in DG_{old}, then we execute it unconditionally. Note that no existing task in DG_{old} can depend or be affected by the new task t.

If task t existed before in DG_{old}, we only execute it if it is actually affected. Since the caller of t awaits the output of t, we use function requireNow to force its checking and possible execution *now*. Task t is affected if it

occurs in queue T_q or if any of its dependencies occurring in T_q will affect it later. Function requireNow repeatedly finds dependency t_{min} of t that is lowest in the dependency graph (closes to the leaves). Since the queue only contains affected tasks, we execute t_{min} and schedule tasks affected by it. We continue until either we have executed the required task t, or until no more dependencies of task t are affected and we can reuse t's output value from the old dependency graph with DG_{old}.outputOf(t). Note that this latter case always triggers for tasks scheduled bottom-up by buildWithChangedFiles, because their dependencies cannot occur in T_q anymore.

The validate function incrementally validates the correctness of the new dependency graph after executing a task t. For a dependency graph to be correct, it may not have overlapping generated files, nor any hidden dependencies. If another task t_{gen} generates the same file f as t does, there is an overlapping generated file and execution is aborted. Furthermore, if t requires a file f without a (transitive) task dependency on t_{gen} that generates f, there is a hidden dependency and execution is aborted. In both cases, this signals that there is an error in the build script.

8.4.3 Properties

An incremental build algorithm is correct if it produces the exact same result as a clean build. Therefore, all affected and new tasks must be executed. Our algorithm is correct for tasks in the old dependency graph: if a task is affected, it will be scheduled. A task is affected directly by depending on a changed file, or indirectly (transitively) by depending on a changed file that an affected task generates, or by depending on the changed output of an affected task. All indirectly affected tasks are always found by traversing the dependency graph bottom-up, through polling the queue and scheduling affected tasks. Finally, all scheduled tasks are executed.

Our algorithm is also correct for new tasks that are executed top-down like the Pluto algorithm, which is correct [36]. The only difference is the reqireNow function which first executes the dependencies of the task, but does eventually execute the task itself. Therefore, the hybrid algorithm is correct.

For optimality, we only consider and execute affected tasks. For existing tasks, this is true because only affected tasks are scheduled. New tasks are affected and always executed. However, we only want to execute *needed* tasks. The hybrid algorithm considers all task in the old dependency graph as needed. This is an overapproximation, because it can happen that an affected task is not needed any more after top-down execution, since a task may remove its dependency to an affected task. Therefore, theoretically, the hybrid algorithm is only partially optimal. However, this is a rare case, as shown in the evaluation in section 8.6.

8.5 Implementation

We have implemented the hybrid algorithm as an alternative execution algorithm for PIE [91], a system for developing interactive software development

pipelines, consisting of a DSL and API for implementing interactive pipelines, and a runtime for incrementally executing them. Interactive software development pipelines are similar to incremental build systems: they are used to incrementally build software artifacts, and also require fast feedback for usage in interactive environments with many low-impact changes such as IDEs and code editors. PIE builds forth on Pluto by reusing its model and algorithm, but provides a concise and expressive DSL for developing interactive pipelines and build scripts, minimizing boilerplate in contrast to Pluto's Java API.

Our algorithm is implemented as a separate executor in the PIE runtime, fully conforming to its API. That is, we can run existing PIE build scripts without changes to our algorithm. Furthermore, since PIE implements the Pluto build algorithm, we can compare our algorithm against Pluto's, for the exact same build scripts. PIE, including our hybrid algorithm, is open source software that can be found online [116].

8.6 Evaluation

In this section, we evaluate the performance of the hybrid algorithm, compared to Pluto's pure top-down algorithm. We describe our experimental setup, show the results, interpret them, and discuss threats to validity.

8.6.1 Experimental Setup

We compare the performance of the Pluto incremental build algorithm, as implemented in the PIE runtime, against our hybrid incremental build algorithm, which we have implemented in PIE runtime.

Build Script. As a build script, we reuse the Spoofax-PIE pipeline, a reimplementation of a large part of the Spoofax pipeline, which was used as a case study of PIE [91] (section 7.6, listing 7.5). Spoofax [75] is a language workbench [38] (a set of tools for developing languages) in which languages are specified in terms of meta-languages, such as SDF [153] for syntax specification, and NaBL [5, 109, 89] for name and type analysis. The Spoofax pipeline derives artifacts from a language specification, such as a parse table for parsing, and a constraint generator and solver for solving name and type analysis. Furthermore, Spoofax supports interactive language development in an IDE setting, enabling a language developer to modify a language specification, resulting in immediate feedback in example programs of that language, and also supports developing multiple languages side-by-side. The Spoofax-PIE reimplementation supports these features. The build script is open-source and can be found online [132].

As input, the Spoofax build script takes a workspace directory consisting of language specifications, where each language specification has a configuration file describing how to build the language specification, a specification of the syntax, styling, and name and type analysis in meta-languages, and example programs. A configuration file at the root of the workspace lists the locations of all language specifications, and locations of the Spoofax meta-languages. As a concrete workspace, we use a directory with three Spoofax language

specifications for the Tiger, Calc, and MiniJava languages.

Describing the Spoofax build script is outside of the scope of this paper. However, we do argue why Spoofax requires a programmatic build script with dynamic dependencies. The Spoofax build script frequently makes use of conditional building, where the result of executing a task influences a condition for another task. For example, when a program fails to parse, the program cannot be analyzed, since analysis requires an AST. Therefore, a condition that checks whether the parsing task succeeds guards the analysis task. Furthermore, Spoofax also makes frequent use of iterative building, where tasks are invoked multiples on different inputs which are outputs of previous tasks. For example, there is a single task description for parsing a file, which is dispatched based on the result of parsing the workspace configuration file, parsing the language specification configuration files, the concrete files that are in the workspace, and the extension of each file. Without a programmatic build script, all these forms of variability would have to be encoded in the configuration step of a declarative build script, which is not possible because many values only become evident during build script execution.

Changes. To measure incremental performance, we have synthesized a chain of 60 realistic changes with varying impacts. First, a from-scratch build is performed that builds all language specifications. Then, we make changes in the form of opening or changing a text editor, requiring execution of a task that provides feedback for that editor, or of modifying and saving a file, which requires execution of tasks that keep the workspace up-to-date.

Changes include: editing and saving example programs, styling specifications, syntax specifications, and name and type analysis specifications; adding a new language specification; undoing changes; and two extreme cases where we run the build with no or all files changed. These changes have varying impacts, where the impact is determined by how many tasks are affected by a change, and the run time of those tasks. For example, changing a syntax definition file requires recompilation of the parse table and reparsing of all example programs. Changing the name and types specification has a larger impact because it requires regeneration of a constraint generator, compilation of the constraint generator, application of the generator against all example programs, and finally application of the constraint solver to solve all generated constraints. A small impact change is editing an example program in an editor, which just requires parsing, styling, and analysis for that program.

We run the exact same changes against the Pluto and our hybrid algorithm, with the only difference that we pass the changed files to our hybrid algorithm, whereas Pluto does not require this. We run the chain of changes against one algorithm in one go, to simulate a full editing session.

Technicalities. We run the benchmark using the JMH [69] benchmarking framework, which runs the benchmark for an algorithm in a separate forked JVM, letting the JVM JIT fully specialize to that algorithm. Furthermore, it runs the benchmark multiple times before starting measurements, to ensure that the JVM is warmed up. Finally, it ensures that the garbage collector is executed

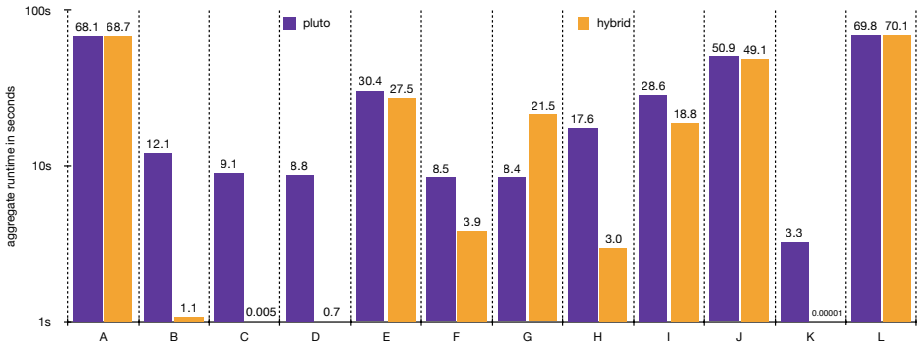


Figure 8.2: Column chart with aggregate benchmark time measurements. The x-axis represent the different changes (described below), the y-axis represents the time taken in seconds in logarithmic scale. For each change, we show the time taken for the Pluto algorithm and our hybrid algorithm. A = initial build, B = all editor changes, C = example program changes, D = styling specification changes, E = adding language specification, F = syntax specification small change, G = syntax specification cascading change, H = syntax specification refactor, I = analysis specification changes, J = analysis specification refactor, K = no changes, L = all files changed.

before running a benchmark, so that garbage produced in a previous run does not influence the new one.

We have executed the benchmark on a MacBook Pro with a 2.7 GHz Intel Core i7 processor, 16 GB of 1600 MHz DDR3 memory, and a SSD, running macOS 10.12.6. The benchmark was executed with a 64-bit JRE of version 1.8.0b144, with 16 MB of stack memory, and 4 GB of heap memory.

8.6.2 Results and Interpretation

We now show the benchmarking results and interpret them. It is not possible to discuss time measurements for all 60 changes. Therefore we aggregate the time taken for different kinds of changes and show those instead. Figure 8.2 shows the time measurements for each aggregated change, for both the Pluto and hybrid algorithm, in a column chart with logarithmic scale. We now go over the results for each kind of change.

A) Initial build. First, we perform an initial build, building all language specifications. To obtain the initial dependency graph, we use top-down building. Therefore, both the Pluto and hybrid algorithm perform identically.

B) Editor changes. We aggregate the running time for all editor changes: both opening new editors and editor text changes, for example programs and language specifications. For all editor changes combined, the hybrid algorithm is 11 seconds faster, providing a speedup of 1016%. The speedup is high because these changes have a small impact, and therefore are efficiently handled by the hybrid algorithm. It is important to quickly process editor

changes in IDEs, as programmers make many changes to editors and require fast feedback cycles.

C) Example program file changes. We modify the files of several example program, and add a new example program file. For these changes combined, the hybrid algorithm takes 0.005 seconds, providing a 9 second (182133%) speedup. Again, these changes have a small impact, and are therefore efficiently handled by the hybrid algorithm, whereas the Pluto algorithm still requires checking of the entire dependency graph.

D) Styling specification change. We modify the styling specification of the Calc language, and add a styling specification to the Tiger language. For these changes, the hybrid algorithm is 8 seconds faster, providing a 1245% speedup. The impact of these changes are slightly larger: changes to the styling specification require re-styling of open editors, but are still relatively smaller in impact and thus efficiently handled.

E) Adding language specification. We add the MiniJava language to the workspace, requiring it to be built, and its example programs to be processed. Since this change causes many new tasks to be executed, its impact is large. The hybrid algorithm performs roughly the same as the Pluto algorithm, providing only a 2.9 second (11%) speedup because of reduced dependency graph checking.

F) Syntax specification small change. We modify lexical syntax definition of the Calc language to parse numbers incorrectly, and undo the change afterwards. This requires the parse table to be rebuilt, and requires processing of Calc's example programs. The hybrid algorithm provides a 4.5 second (118%) speedup, because a smaller part of the dependency graph is checked.

G) Syntax specification cascading change. We modify the Calc syntax definition in such a way that the resulting parser will fail to parse all example programs, and also in such a way that new AST signatures need to be generated. From the syntax specification, Spoofax generates AST signature files that the name and type analysis uses. These signature files have changed, therefore requiring the name and type analysis specification to be recompiled. Finally, all example programs must be reparsed and reanalyzed.

However, because example programs cannot be parsed any more, they also cannot be analyzed any more, since name and type analysis requires an AST. Therefore, the dependency from the process example file task, to the task that analyzes the AST of an example program, disappears. The Pluto algorithm first visits the process example file task, which removes its dependency to the analysis task, and therefore never recompiles the name and type analysis specification. However, the hybrid algorithm goes bottom-up to first recompile the name and type analysis specification, and only then executes the process example file task, therefore executing a task that was not required to be executed. In this case, the hybrid algorithm was 13 seconds slower, causing a 61% slowdown.

This is a tradeoff of the hybrid algorithm: if a dependency to a task disap-

pears, the hybrid algorithm will still visit it. However, these cases are very rare, only a single change triggers this kind of behavior. For example, if at least one example program could be parsed into an AST (possibly through error recovery), the analysis specification has to be recompiled. We undo the change afterwards to make example programs parse again.

H) Syntax specification refactor. We refactor a part of the MiniJava syntax definition into another file, which results in a semantically equivalent parser. The hybrid algorithm provides a 14.5 second (483%) speedup, because it first rebuilds the parse table, detects that it did not change, and then cuts off the build early. Contrary to the previous change, a bottom-up traversal here helps in cutting down the incremental build time, by not even traversing the unaffected part of the dependency graph.

I) Analysis specification change. We modify the name and type analysis specification of the Calc language, such that it scopes bindings differently, and undo the change afterwards. Because changing these specifications has a moderate impact, the hybrid algorithm provides a moderate 9.7 second speedup (52%).

J) Analysis specification refactor. We refactor a part of the Tiger name and type analysis specification into another file. Even though this results in a semantically equivalent analyzer, the change detection of the Spoofax-PIE build script is not smart enough to detect this. Because compiling the name and type analysis specification, and then performing constraint solving for all Tiger example programs, is expensive, this change has a large impact. Therefore, the Pluto and hybrid algorithm perform nearly identically.

K) No changes. When there are no changes, the hybrid algorithm essentially performs no work, completing in sub-millisecond time, while the Pluto algorithm still needs to check the entire dependency graph, costing 3.3 seconds of time. This is the constant overhead that even small-impact changes suffer from with the Pluto algorithm, which the hybrid algorithm saves.

L) All files changed. Finally, we change all source files by appending a space to the end of each file. Realistically, this kind of change can happen when checking out a different branch in a source control management system such as Git. When all source files change, using a bottom-up approach makes no sense, since (almost all) tasks will be affected, while incurring overhead because of scheduling. Therefore, we detect when more than 50% of source files (all required files, for which there is no generator task) change, and run a top-down build with the Pluto algorithm instead, therefore running as fast as the Pluto algorithm does. This heuristic seems to work well, but may require further tweaking.

Conclusion. We can conclude that, for this build script and workspace directory, our algorithm scales better with the impact of a change than the Pluto algorithm, for many kinds of changes. The only exceptions being when all files are changed, for which a full rebuild could be triggered, or when a dependency to an expensive task is removed, which rarely happens.

8.6.3 Threats to Validity

A possible threat to validity is that we have benchmarked the algorithms against a single build script. However, it is a complex build script that represents the realistic scenario of interactive language development in a language workbench. For example, the build script requires dynamic file dependencies in order to track precise dependencies which only become evident during a build. Furthermore, it also requires dynamic task dependencies, in order to dispatch the correct tasks based on the configuration of the workspace and each language specification.

Another possible threat is that we have synthesized a chain of changes, instead of using existing change scenarios. However, we have constructed 60 changes to different kinds of aspects; such as changing an example program, and changing a file of the syntax specification; and with varying levels of impact, ranging from changing the text in a single editor, to changing a file of the name and type specification, which transitively affects many other tasks.

8.7 Related Work

We now discuss related work, starting with the large body of work on incremental build systems with static dependencies.

Static Dependencies. Make [133] is an incremental build system with declarative build rules based on files. It has limited support for dynamic file dependencies, and no proper support for dynamic task dependencies. Because of these limitations, Make scales well for simple build scripts, since it can first topologically sort dependencies, iterate over the dependencies, and incrementally execute affected tasks. While it is possible to emulate dynamic task dependencies, this requires tedious Makefile generation, encoding of all dependencies as files, and recursive Make execution. Therefore, Make is not sufficient for more complicated build scripts.

Many build systems follow a similar approach to Make, first building a task DAG, and then executing it. For example, Gradle [67], Bazel [47], Buck [40], PROM [79], Fabricate [64], Tup [127], and Ninja [99] follow this approach, with slight variations. Gradle is a build automation tool, programmable in the Groovy language, that, like our hybrid algorithm, also supports values as inputs and outputs of tasks. PROM replaces declarative make rules with logical programming, while keeping the same incremental build algorithm. Fabricate uses system tracing to automatically infer file dependencies, but is only supported on Linux. Tup, like our hybrid build algorithm, requires a list of changed files as input, instead of scanning all files, to more efficiently build the task DAG. Ninja, unlike Make, detects changes to the commands of a rule, resulting in a rebuild if the rule is changed. None of the above systems supports dynamic file or task dependencies.

Dynamic Dependencies. Some build systems intertwine incremental execution with the discovery of file and task dependencies. Pluto [36] is a Java library for developing incremental build scripts with dynamic dependencies. As

discussed throughout this paper, Pluto uses a top-down algorithm that does not scale to small changes over large dependency graphs.

OMake [62] is a build system with Make-like syntax, but with a richer dependency tracking mechanism and a more complicated algorithm. It has limited support for dynamic file dependencies through scanner rules that scan depfiles and register their dependencies during execution. However, it does not support dynamic task dependencies; all tasks dependencies are specified statically in the build rules.

Shake [107, 108] is a Haskell library for implementing build scripts with incremental execution. It has limited support for dynamic file dependencies, allowing needed files to be discovered dynamically, but generated file dependencies must be specified statically as the build target. It also has limited support for dynamic task dependencies: Tasks are named by keys, and those tasks can be required like files through their keys. However, these tasks are not parameterized, nor can they return values, making their use as dynamic task dependencies tedious.

Incremental Computing. Our work is also related to approaches on incremental computing. Datalog [22] is a logic programming language with incremental solvers [50]. The are several differences between our hybrid algorithm and incremental Datalog solvers. Datalog solvers can deal with cycles, eagerly compute all facts, and use static dependencies from the Datalog program, whereas the hybrid algorithm (and build systems in general) disallow cycles, only compute demanded facts, and use dynamic dependencies.

Adapton [52, 51] is a library for on-demand (lazy) incremental computation. Like the Pluto and our hybrid algorithm, Adapton supports a form of dynamic task dependencies: dynamic computation dependencies which form a computation (thunk) graph. Initially, Adapton builds a full computation graph. When the computation graph is affected by a change, edges of thunks are marked as dirty. Then, when a dirtied computation is demanded, it transitively "cleans" the edges of thunks during change propagation, re-executing out-of-date computations. Nominal Adapton's [51] formal development relaxes the ordering of dirtying and cleaning, allowing dirtying while change propagation is running. However, the algorithm or implementation with this relaxed ordering is not described.

8.8 Conclusion

We have shown the need for an efficient, precise, and expressive build system. Many build systems are efficient and precise, but not expressive, making complex build script development tedious. Pluto, a recent incremental build system that supports programmable build scripts with dynamic dependencies, is expressive, but does not scale with the impact of a change, because it requires a top-down traversal over the entire dependency graph for each change. To overcome this scalability problem, we have realized a hybrid algorithm that mixes bottom-up building for scalability, and top-down building for expressiveness through dynamic dependencies. We have evaluated the

performance of our hybrid algorithm against Pluto's algorithm, with a case study on the Spoofax language workbench. The evaluation demonstrates that the hybrid algorithm, with the exception of one kind of change, indeed scales better with the impact of a change, and is therefore faster than the Pluto algorithm, in particular for low-impact changes.

Acknowledgments

This research was supported by NWO/EW Free Competition Project 612.001.114 (Deep Integration of Domain-Specific Languages) and NWO VICI Project (639.023.206) (Language Designer's Workbench).

Conclusion

9

We now conclude by summarizing interactive programming systems, our vision of language-parametric methods, and our five core contributions. We discuss our thesis, and how our core contributions relate to it. Finally, we end by discussing directions for future work.

9.1 Interactive Programming Systems

A *programming system* for a programming language supports the development of programs through a batch compiler that validates programs and by transforming those programs into executable forms. An *interactive programming system* additionally supports the development of programs by providing automatic, continuous, responsive, and inline feedback to the programmer. However, a programming system is only truly interactive when it is *correct* and *responsive*.

A method to achieve responsiveness is *incrementality*, where the response time is proportional to the impact of a change. Furthermore, responsiveness is only achieved when incrementality is *scalable*, where the incremental system can *scale down* to many low-impact changes, and *scale up* to large inputs. This is especially important in interactive programming systems, since the majority of changes are small (e.g., typing a character into a source code editor), and programs are large. Finally, it is important that correctness is still guaranteed in the presence of scalable incrementality.

However, manually implementing an incremental system is a challenge as it requires the application of cross-cutting techniques such as dependency tracking, caching, cache invalidation, change detection, and persistence, which are complicated and error-prone to implement. Furthermore, scalable incrementality increases the challenge, as incrementality must scale up to large dependency graphs, cache large amounts of data, do cache invalidation through these large graphs, and detect low-impact changes. Finally, the scalable and incremental implementation must be correct, which is unlikely when manually implementing complicated and error-prone techniques.

Therefore, our vision is to use *language-parametric methods* to develop responsive and correct interactive programming systems. Such a language-parametric method takes as input an implementation or description of a programming language, and automatically produces (parts of) an interactive programming system, without the language developer having to manually implement a correct, incremental, and scalable interactive programming system.

Our thesis is that these *language-parametric methods for developing interactive programming systems are feasible and useful*. We have shown feasibility of language-parametric methods in the five core contributions of this dissertation by developing three language-parametric methods: declarative specification of

incremental name and type analysis, bootstrapping of language workbench meta-DSLs, and pipelining of interactive programming systems.

9.2 Language-Parametric Methods

We now summarize the language-language parametric methods developed in this dissertation and discuss their usefulness.

9.2.1 Incremental Name and Type Analysis

We have developed a language-parametric method for incremental name and type analysis consisting of the NaBL (chapter 2), and the incremental task engine (chapter 3).

NaBL is a meta-DSL for declarative specification of name binding and scope rules of programming languages in terms of definitions and use sites, properties of names, namespaces for separating categories of names, scopes in which definitions are visible, and imports between scopes. From such a specification, the NaBL compiler derives a name analysis and editor services for inline error checking, reference resolution, and code completion, freeing the language developer from having to manually implement these parts. Therefore, NaBL is a language-parametric method for developing correct name analysis and corresponding interactive editor services.

We extend NaBL with incrementality and type checking, using a language independent task engine for incremental name and type analysis. In this approach, we specify name and scope rules in NaBL, typing rules in TS – a meta-DSL for simple type system specification – from which we automatically derive a traversal that collects naming and typing tasks when given a program. These tasks are sent to the task engine, which then executes changed tasks to incrementally execute name and type analysis, updating data structures required for editor services, and responsively updating inline error messages. Therefore, NaBL, TS, and the task engine are a language-parametric method for developing correct and responsive name and type analysis with corresponding editor services.

We have evaluated NaBL by specifying the name binding of a subset of C# (sections 2.2 and 2.3), and have evaluated the task engine approach by specifying the name and type rules of the WebDSL language and by confirming responsiveness through benchmarking (section 3.6). Furthermore, as discussed in chapter 4, the task engine approach is used to specify and run the incremental name and type analysis of the Green-Marl [63, 94] DSL, and the NaBL, TS, and SDF [153] meta-DSLs of the Spoofax [75] language workbench. Finally, NaBL, TS, and the task engine were used to teach students about incremental name and type analysis as part of the 2015-2016 edition of the Compiler Construction lab [145] where they develop a full version of the MiniJava [9] language.

9.2.2 *Bootstrapping meta-DSLs of Language Workbenches*

A bootstrapped compiler can compile its own source code, because the compiler is written in the compiled language itself, providing several benefits such as a high-level implementation of the compiler, a large-scale test case for the compiler, and improvement dissemination. However, DSLs have limited expressiveness (by design) and are therefore ill-suited for bootstrapping. Therefore, language workbenches provide high level meta-languages for developing DSLs and their compilers, freeing language developers from having to bootstrap their DSLs. What we desire instead, is bootstrapping the meta-language compilers of language workbenches, to inherit the benefits of bootstrapping stated above.

However, bootstrapping a language workbench is complicated by the fact that most provide *multiple separate domain-specific meta-languages* for describing different language aspects such as syntax, name and type analysis, code generation, and so forth. Thus, in order to build a meta-language compiler, we need to apply multiple meta-language compilers, entailing intricate dependencies that sound language workbench bootstrapping needs to handle. Furthermore, meta-languages often *generate generators*, which may in turn generate more generators, requiring an unknown number of build iterations to apply all generators and possibly find defects in them.

Our solution to these problems is to do versioning and dependency tracking between meta-languages, and perform *fixpoint bootstrapping*, where we iteratively self-apply meta-language compilers to derive new versions until no change occurs, or until we find a defect (listing 5.1). Bootstrapping operations can be started, cancelled (when diverging), and rolled back (when defect) interactively, supporting the interactive programming system of the language workbench. In conclusion, our bootstrapping approach provides a (meta)language-parametric method for correctly and interactively bootstrapping the meta-languages of language workbenches, in an interactive programming environment.

We have evaluated our bootstrapping approach by bootstrapping the eight meta-DSLs of the Spoofax language workbench (section 8.6). We make seven realistic changes to one or more meta-languages and perform fixpoint bootstrapping operations. We were able to create new baselines after successful bootstrapping attempts; make breaking changes by decomposing changes into multiple compatible ones; find defects in changes, roll back to the existing baseline, fix the defect, and reattempt bootstrapping; and perform these operations in the interactive programming system of Spoofax.

To this day, we are still bootstrapping the meta-languages of Spoofax as part of its build. We are still versioning, creating explicit dependencies, and releasing new baselines of Spoofax's meta-languages, even with the addition of two extra meta-DSLs: FlowSpec [129] and Statix [6].

9.2.3 *Pipelining of Interactive Programming Systems*

We have developed PIE, which provides a language-parametric method for developing interactive software development pipelines, a superset of correct and

responsive interactive programming environments (chapter 7); and developed a change-driven algorithm for PIE which makes it scalable (chapter 8).

An interactive software development pipeline automates parts of the software engineering process, such as building software via build scripts, but also reacts immediately to changes in input, and provides timely feedback to the user. An interactive programming system is an instance of such a pipeline, where changes to programs are immediately processed to provide timely feedback to programmers. However, interactivity complicates the development of pipelines, if responsiveness and correctness become the responsibility of the pipeline programmer, rather than being supported by the underlying system.

PIE consists of a DSL, API, and runtime for developing correct and responsive interactive software development pipelines, where ease of development *is* a focus. The PIE DSL serves as a front-end for developing pipelines with minimal boilerplate in a functional language. The PIE API is a lower-level front-end for developing foreign pipeline functions which cannot be modeled in the DSL. Finally, the runtime incrementally executes pipelines implemented in the API using Pluto's incremental build algorithm.

However, the incremental build algorithm that we used did not scale, because it needs to traverse the entire dependency graph (produced in a previous build) from top to bottom, making the run-time of the algorithm dependent on the size of the dependency graph, *not* the impact of the change. This quickly becane a problem in interactive programming systems, where there are many changes and those changes have a low-impact (e.g., programmer typing characters into an editor), while the program and its induced dependency graph is large.

To solve this scalability problem, we have developed a new incremental build algorithm that performs change-driven rebuilding (listings 8.2a to 8.2c). Our algorithm scales with the impact of a change, and is independent from the size of the dependency graph, because it only ever visits affected tasks. Therefore, PIE with our change-driven incremental build algorithm provides a language-parametric method for developing correct and responsive interactive programming systems. Furthermore, PIE can more generally be applied to interactive software development pipelines, such as build scripts, continuous integration pipelines, benchmarking pipelines.

We evaluate PIE with a case study by reimplementing a significant part of the interactive programming system of the Spoofax language workbench (listing 7.5). The existing pipeline of Spoofax's interactive programming system was scattered across four different formalisms, decreasing ease of development; overapproximates dependencies, causing loss of incrementality; and underapproximates dependencies, causing loss of correctness. However, with PIE, we can easily integrate the different components of Spoofax; such as its parser, analyzers, transformations, build scripts, editor services, meta-languages, and dynamic language loading; into a single formalism. PIE ensures that the pipeline is correct and responsive, without the pipeline programmer having to implement techniques such as incrementality, or without having to reason about correctness.

We also experimentally evaluate the performance of our change-driven

bottom-up algorithm with the Spoofax pipeline (fig. 8.2). To measure incremental performance an scalability, we synthesized a chain of 60 realistic changes of varying types and impacts, ranging from changing an example program, to changing the syntax specification of a language, to adding a new language specification. Results show that for low-impact changes (i.e., changes that only cause a small number of tasks to be actually affected), our change-driven algorithm is several orders of magnitude faster than the previous algorithm we used, while not slower for high-impact changes.

Finally, to show that PIE can be used for other interactive software development pipelines, we have also performed a case study with the benchmarking suite from the accompanying artifact [136] of Criterion [137], which measures performance of immutable data structures on the JVM. Criterion uses a bash script to orchestrate benchmarking tasks, requiring to re-run all benchmarks after changes. We converted this script into a PIE pipeline (listing 7.6), which runs each benchmark and subject pair in isolation, enabling incrementality where PIE only executes a benchmark against all subjects if a benchmark changes, and only executes a subject against all benchmarks when a subject changes.

9.3 Future Work

We now discuss future work on incremental name and type analysis, bootstrapping of meta-DSLs, and pipelining of interactive programming systems.

9.3.1 Incremental Name and Type Analysis

Expressiveness. As discussed in chapter 4, future work for NaBL, TS and the task engine is to increase the expressiveness of the approach to support more kinds of names, scopes, and type systems. Much of this future work has already been done in the context of Scope Graphs [109], an extension of scope graphs into a full analysis framework based on constraint solving [5], and further improvements to expressiveness [6]. However, making this constraint-based analysis framework feasible for interactive programming systems is still a challenge, as it is not yet incremental nor scalable, and does not provide editor services such as good inline error messages, code completion, semantic code styling, and occurrence highlighting. Therefore, future work should focus on the combination of high expressiveness of name and type systems, while properly supporting correct and responsive interactive programming systems.

Type-Directed Transformations. Furthermore, transformations are frequently name or type-directed instead of syntax-directed (e.g., compile a Java class into a class file, for every public Java class in the program), or need to query (properties of) names or types. NaBL supports this by building an index of names and providing an API to it, and the task engine supports this by building tasks which represent queries, for which the value is available through an API when performing a transformation. While it is possible to do name and type-directed transformations, these APIs feel ad-hoc, and do not support incremental transformations. One interesting direction of future work is to fig-

ure out a high-quality API for name and type-directed transformations, name and type queries, and how to (automatically) incrementalize transformations with such an API.

Updating Analysis Data after Transformations. Finally, a frequently occurring pattern in compilation is to perform multiple small optimization transformations to the program, where each one transforms the program into a new (semantically equivalent) program. However, after such a transformation, the name and type information can be invalidated, because names or types may have been renamed, created, removed, or moved into a different scope. Since optimizations need access to name and type information, we must re-execute name and type analysis to make that information up-to-date again. Even when name and type analysis is incremental, it is costly to do so because of change detection and other overhead, which dominates when hundreds or even thousands of small optimization transformations occur. Therefore, another interesting direction of future work is to figure out how to incrementally update name and type information after (small) transformations.

9.3.2 Bootstrapping of Meta-DSLs

Our fixpoint bootstrapping approach either produces a new baseline or finds a defect after a number of fixpoint iterations, or diverges when the compilers of the meta-DSLs diverge. Therefore, our approach is a dynamic one: we can only figure this out after executing the compilers of the meta-DSLs. Future work could be to find out a way to do this statically, possibly by exploiting the fact that compilers typically converge.

Finally, bootstrapping can also be seen as a sort of pipeline. It is currently not possible to perform fixpoint bootstrapping with PIE, as PIE does not have a fixpoint operations. If we want to integrate fixpoint bootstrapping in PIE, we need to add an incremental fixpoint operation to PIE.

9.3.3 Pipelining of Interactive Programming Systems

There is a lot of future work for PIE in the space of pipelining of interactive programming systems, and software development pipelines in general.

Observability. PIE currently does not track if (the effect of) a task is observable to the outside world. For example, if we create a task that provides feedback for a code editor (e.g., code styling and inline error messages), but that code editor is currently not visible (e.g., it is hidden behind another window or was closed by the programmer), then we do not need to execute that task when the code changes, because the effect is not observable. We need to track which tasks are observable to the outside world, provide operations for marking tasks as (un)observed, and never execute tasks that are directly or transitively unobserved, to increase efficiency. The challenge is to find an efficient way to maintain this information, while keeping the incremental build algorithm correct. Furthermore, observability information could be used to perform garbage collection of tasks that are no longer required.

Concurrency. PIE currently does not support concurrency or parallelism. Interactive programming systems are concurrent systems where multiple things can happen concurrently, such as running name and type analysis on one program, while parsing another program, while the user is requesting code completion. Similarly, processors have many cores which require parallelism to exploit. Concurrent and parallelism support is a challenge in the presence of dynamic dependencies, as multiple tasks could read/write to files concurrently without prior knowledge, causing conflicts. Furthermore, concurrently running tasks require a consistency model such as eventual consistency, as tasks may use the values produced by other tasks. Extending the PIE model and algorithm with concurrency and parallelism while overcoming these problems is a challenge. It is also a technical challenge, as efficient concurrent and parallel execution is hard to implement right, especially in the presence of the mutable filesystem, incrementality, and persistence.

Deferred Tasks. PIE currently does not support deferring a task while it is executing: a task either fully executes, or fails. Deferring execution is useful in the case where a task currently does not have enough information yet to execute, and that information cannot be retrieved by executing another task, because it is unknown which task provides this information. For example, in name analysis, use sites in a module frequently refer to definition sites in other modules through imports. However, when the other module has not been analyzed yet, and it is unknown where this module resides, it is not possible to complete name analysis for the current module, and the task must be deferred until the other module has been analyzed. The challenge here is to extend the PIE model and algorithm with support for deferring tasks in an efficient way, possibly through coroutines or other asynchronous programming models.

Partial Evaluation. Partial evaluation could be used to automate deployment in PIE pipelines. Tasks in a pipeline depend on other data by depending on the files or output values from other tasks. Sometimes, this data is completely dynamic. For example, in a live language development pipeline for a DSL, the task that compiles programs of the DSL depends on another task that builds the compiler, which in turn depends on the compiler specification source files of the DSL. Whenever this compiler specification changes, a new compiler is built, and all example programs are recompiled with this new compiler.

On the other hand, when we want to deploy the DSL to a customer's computer, the compiler does not change any more and becomes completely static. Instead of deploying the compiler specification source files to the customer, we would rather only deploy the compiler, to avoid the customer having to build the compiler, having to store source files which never change, and possibly to prevent reverse engineering of the compiler via source code. Currently, to achieve this, we would need to manually adapt the pipeline to accept both compiler specification source files and a built compiler, which is tedious. With partial evaluation, we can automate this process by specifying which input data is static, execute the tasks of the pipeline, and replace tasks which (transitively) depend on completely static data with a task that just

returns the static data.

Applications. Finally, we want to keep evaluating PIE by application to more subdomains of interactive software development pipelines.

Bibliography

[1] Wil M. P. van der Aalst and Arthur H. M. ter Hofstede. "YAWL: yet another workflow language". In: *Inf. Syst.* 30.4 (2005), pp. 245–275. DOI: 10.1016/j.is.2004.02.002.

[2] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, Aug. 2006.

[3] Johan Åkesson, Torbjörn Ekman, and Görel Hedin. "Implementation of a Modelica compiler using JastAdd attribute grammars". In: *Science of Computer Programming* 75.1-2 (2010), pp. 21–38. DOI: 10.1016/j.scico.2009.07.003.

[4] Peter Amstutz, Michael R. Crusoe, Nebojša Tijanić, Brad Chapman, John Chilton, Michael Heuer, Andrey Kartashov, Dan Leehr, Hervé Ménager, Maya Nedeljkovich, Matt Scales, Stian Soiland-Reyes, and Luka Stojanovic. "Common Workflow Language, v1.0". In: (2016). DOI: 10.6084/m9.figshare.3115156.v2.

[5] Hendrik van Antwerpen, Pierre Néron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. "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, PEPM 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. Ed. by Martin Erwig and Tiark Rompf. ACM, 2016, pp. 49–60. ISBN: 978-1-4503-4097-7. DOI: 10.1145/2847538.2847543.

[6] Hendrik van Antwerpen, Casper Bach Poulsen, Arjen Rouvoet, and Eelco Visser. "Scopes as types". In: *Proceedings of the ACM on Programming Languages* 2.OOPSLA (2018). DOI: 10.1145/3276484.

[7] Apache. *Spark*. https://spark.apache.org/. (Visited on 10/04/2019).

[8] Andrew W. Appel. "Axiomatic Bootstrapping: A Guide for Compiler Hackers". In: *ACM Transactions on Programming Languages and Systems* 16.6 (1994), pp. 1699–1718. DOI: 10.1145/197320.197336.

[9] Andrew W. Appel. *Modern Compiler Implementation in Java, 2nd edition*. Cambridge University Press, 2002. ISBN: 0-521-82060-X.

[10] John Warner Backus. "The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference". In: *IFIP Congress*. 1959, pp. 125–131.

[11] Jon Bentley. "Programming pearls: little languages". In: *Commun. ACM* 29 (1986). DOI: 10.1145/6424.315691.

[12] Lorenzo Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. 2nd. Packt Publishing, 2016.

[13] Mark G. J. van den Brand. “PREGMATIC - a generator for incremental programming environments”. PhD thesis. University Nijmegen, 1992.

[14] Mark G. J. van den Brand, Arie van Deursen, Jan Heering, H. A. de Jong, Merijn de Jonge, Tobias Kuipers, Paul Klint, Leon Moonen, Pieter A. Olivier, Jeroen Scheerder, Jurgen J. Vinju, Eelco Visser, and Joost Visser. “The ASF+SDF Meta-environment: A Component-Based Language Development Environment”. In: *Compiler Construction, 10th International Conference, CC 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings*. Ed. by Reinhard Wilhelm. Vol. 2027. Lecture Notes in Computer Science. Springer, 2001, pp. 365–370. ISBN: 3-540-41861-X. DOI: 10.1016/S1571-0661(04)80917-4.

[15] Mark G. J. van den Brand, Jeroen Scheerder, Jurgen J. Vinju, and Eelco Visser. “Disambiguation Filters for Scannerless Generalized LR Parsers”. In: *Compiler Construction, 11th International Conference, CC 2002, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*. Ed. by R. Nigel Horspool. Vol. 2304. Lecture Notes in Computer Science. Springer, 2002, pp. 143–158. ISBN: 3-540-43369-4. DOI: 10.1007/3-540-45937-5_12.

[16] Harvey Bratman. “A alternate form of the "UNCOL diagram"”. In: *Communications of the ACM* 4.3 (1961), p. 142. DOI: 10.1145/366199.366249.

[17] Martin Bravenboer, Arthur van Dam, Karina Olmos, and Eelco Visser. “Program Transformation with Scoped Dynamic Rewrite Rules”. In: *Fundamenta Informaticae* 69.1-2 (2006). https://content.iospress.com/articles/fundamenta-informaticae/fi69-1-2-06, pp. 123–178.

[18] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. “Stratego/XT 0.17. A language and toolset for program transformation”. In: *Science of Computer Programming* 72.1-2 (2008), pp. 52–70. DOI: 10.1016/j.scico.2007.11.003.

[19] Doug Brown, John Levine, and Tony Mason. *Lex & Yacc*. 2nd. O’Reilly Series. O’Reilly Media, 1992. ISBN: 9781565920002.

[20] Christoff Bürger, Sven Karol, Christian Wende, and Uwe Aßmann. “Reference Attribute Grammars for Metamodel Semantics”. In: *Software Language Engineering - Third International Conference, SLE 2010, Eindhoven, The Netherlands, October 12-13, 2010, Revised Selected Papers*. Ed. by Brian A. Malloy, Steffen Staab, and Mark van den Brand. Vol. 6563. Lecture Notes in Computer Science. Springer, 2010, pp. 22–41. ISBN: 978-3-642-19439-9. DOI: 10.1007/978-3-642-19440-5_3.

[21] Cristiano Calcagno, Walid Taha, Liwen Huang, and Xavier Leroy. "Implementing Multi-stage Languages Using ASTs, Gensym, and Reflection". In: *Generative Programming and Component Engineering, Second International Conference, GPCE 2003, Erfurt, Germany, September 22-25, 2003, Proceedings*. Ed. by Frank Pfenning and Yannis Smaragdakis. Vol. 2830. Lecture Notes in Computer Science. Springer, 2003, pp. 57–76. ISBN: 3-540-20102-5. DOI: 10.1007/978-3-540-39815-8_4.

[22] Stefano Ceri, Georg Gottlob, and Letizia Tanca. "What you Always Wanted to Know About Datalog (And Never Dared to Ask)". In: *IEEE Trans. Knowl. Data Eng.* 1.1 (1989), pp. 146–166. DOI: 10.1109/69.43410.

[23] Donald D. Chamberlin and Raymond F. Boyce. "SEQUEL: A Structured English Query Language". In: *Proceedings of 1974 ACM-SIGMOD Workshop on Data Description, Access and Control, Ann Arbor, Michigan, May 1-3, 1974, 2 Volumes*. Ed. by Randall Rustin. ACM, 1974, pp. 249–264. DOI: 10.1145/800296.811515.

[24] Philippe Charles, Robert M. Fuhrer, and Stanley M. Sutton Jr. "IMP: a meta-tooling platform for creating language-specific ides in eclipse". In: *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), November 5-9, 2007, Atlanta, Georgia, USA*. Ed. by R. E. Kurt Stirewalt, Alexander Egyed, and Bernd Fischer. ACM, 2007, pp. 485–488. ISBN: 978-1-59593-882-4. DOI: 10.1145/1321631.1321715.

[25] Alan J. Demers, Thomas W. Reps, and Tim Teitelbaum. "Incremental Evaluation for Attribute Grammars with Application to Syntax-Directed Editors". In: *POPL*. 1981, pp. 105–116. DOI: 10.1145/567532.567544.

[26] Arie van Deursen and Paul Klint. "Little languages: little maintenance?" In: *Journal of Software Maintenance* 10.2 (1998). http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.50.4726, pp. 75–92.

[27] Arie van Deursen, Paul Klint, and Joost Visser. "Domain-Specific Languages: An Annotated Bibliography". In: *SIGPLAN Notices* 35.6 (2000), pp. 26–36. DOI: 10.1145/352029.352035.

[28] Eelco Dolstra, Merijn de Jonge, and Eelco Visser. "Nix: A Safe and Policy-Free System for Software Deployment". In: *Proceedings of the 18th Conference on Systems Administration (LISA 2004), Atlanta, USA, November 14-19, 2004*. http://www.usenix.org/publications/library/proceedings/lisa04/tech/dolstra.html. USENIX, 2004, pp. 79–92.

[29] Eelco Dolstra and Andres Löh. "NixOS: a purely functional Linux distribution". In: *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*. Ed. by James Hook and Peter Thiemann. ACM, 2008, pp. 367–378. ISBN: 978-1-59593-919-7. DOI: 10.1145/1411204.1411255.

[30] Eelco Dolstra, Eelco Visser, and Merijn de Jonge. "Imposing a Memory Management Discipline on Software Deployment". In: *26th International Conference on Software Engineering (ICSE 2004), 23-28 May 2004, Edinburgh, United Kingdom*. IEEE Computer Society, 2004, pp. 583–592. ISBN: 0-7695-2163-0. DOI: 10.1109/ICSE.2004.1317480.

[31] Jay Earley and Howard E. Sturgis. "A formalism for translator interactions". In: *Communications of the ACM* 13.10 (1970), pp. 607–617. DOI: 10.1145/355598.362740.

[32] Torbjörn Ekman and Görel Hedin. "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, 2007, pp. 1–18. ISBN: 978-1-59593-786-5. DOI: 10.1145/1297027.1297029.

[33] Torbjörn Ekman and Görel Hedin. "The JastAdd system - modular extensible compiler construction". In: *Science of Computer Programming* 69.1-3 (2007), pp. 14–26. DOI: 10.1016/j.scico.2007.02.003.

[34] EPFL. *The Scala Programming Language*. https://www.scala-lang.org/. (Visited on 10/04/2019).

[35] Sebastian Erdweg. "Extensible Languages for Flexible and Principled Domain Abstraction". PhD thesis. Philipps-Universität Marburg, Mar. 2013.

[36] Sebastian Erdweg, Moritz Lichter, and Manuel Weiel. "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, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*. Ed. by Jonathan Aldrich and Patrick Eugster. ACM, 2015, pp. 89–106. ISBN: 978-1-4503-3689-5. DOI: 10.1145/2814270.2814316.

[37] Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. "SugarJ: library-based syntactic language extensibility". In: *Proceedings of 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, 2011, pp. 391–406. ISBN: 978-1-4503-0940-0. DOI: 10.1145/2048066.2048099.

[38] Sebastian Erdweg, 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. "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, 2013, pp. 197–217. ISBN: 978-3-319-02653-4. DOI: 10.1007/978-3-319-02654-1_11.

[39] Sebastian Erdweg, 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. "Evaluating and comparing language workbenches: Existing results and benchmarks for the future". In: *Computer Languages, Systems & Structures* 44 (2015), pp. 24–47. DOI: 10.1016/j.cl.2015.08.007.

[40] Facebook. *Buck: a fast build tool*. https://buckbuild.com/. (Visited on 10/04/2019).

[41] Robby Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. "DrScheme: a programming environment for Scheme". In: *Journal of Functional Programming* 12.2 (2002), pp. 159–182.

[42] Robby Findler and PLT. *DrRacket: Programming Environment*. Tech. rep. PLT-TR-2010-2. http://racket-lang.org/tr2/. PLT Design Inc., 2010.

[43] Bent Flyvbjerg. "Five Misunderstandings about Case-Study Research". In: *Qualitative Inquiry* 12.2 (Apr. 2006).

[44] Apache Software Foundation. *Ant*. https://ant.apache.org/. (Visited on 10/04/2019).

[45] Apache Software Foundation. *Maven*. https://maven.apache.org/. (Visited on 10/04/2019).

[46] Martin Fowler. *Language Workbenches: The Killer-App for Domain Specific Languages?* http://www.martinfowler.com/articles/languageWorkbench.html. 2005.

[47] Google. *Bazel - a fast, scalable, multi-language and extensible build system*. https://bazel.build/. (Visited on 10/04/2019).

[48] Google. *Guice*. https://github.com/google/guice. (Visited on 10/04/2019).

[49] Danny M. Groenewegen, Zef Hemel, Lennart C. L. Kats, and Eelco Visser. "WebDSL: a domain-specific language for dynamic web applications". In: *Companion to the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008, October 19-13, 2007, Nashville, TN, USA*. Ed. by Gail E. Harris. ACM, 2008, pp. 779–780. ISBN: 978-1-60558-220-7. DOI: 10.1145/1449814.1449858.

[50] Ashish Gupta and Inderpal Singh Mumick. "Maintenance of Materialized Views: Problems, Techniques, and Applications". In: *IEEE Data Eng. Bull.* 18.2 (1995). db/journals/debu/GuptaM95.html, pp. 3–18.

[51] Matthew A. Hammer, Joshua Dunfield, Kyle Headley, Nicholas Labich, Jeffrey S. Foster, Michael W. Hicks, and David Van Horn. "Incremental computation with names". In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*. Ed. by Jonathan Aldrich and Patrick Eugster. ACM, 2015, pp. 748–766. ISBN: 978-1-4503-3689-5. DOI: 10.1145/2814270.2814305.

[52] Matthew A. Hammer, Yit Phang Khoo, Michael Hicks, and Jeffrey S. Foster. "Adapton: composable, demand-driven incremental computation". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*. Ed. by Michael F. P. O'Boyle and Keshav Pingali. ACM, 2014, p. 18. ISBN: 978-1-4503-2784-8. DOI: 10.1145/2594291.2594324.

[53] Görel Hedin. "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, 2009, pp. 166–200. ISBN: 978-3-642-18022-4. DOI: 10.1007/978-3-642-18023-1_4.

[54] Görel Hedin. "Incremental Semantic Analysis". PhD thesis. 1992.

[55] Görel Hedin. "Incremental Static-Semantic Analysis for Object-Oriented Languages Using Door Attribute Grammars". 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, 1991, pp. 374–379. ISBN: 3-540-54572-7.

[56] Görel Hedin. "Reference Attributed Grammars". In: *Informatica (Slovenia)* 24.3 (2000).

[57] Florian Heidenreich, Jendrik Johannes, Sven Karol, Mirko Seifert, and Christian Wende. "Derivation and Refinement of Textual Syntax for Models". In: *Model Driven Architecture - Foundations and Applications, 5th European Conference, ECMDA-FA 2009, Enschede, The Netherlands, June 23-26, 2009. Proceedings*. Ed. by Richard F. Paige, Alan Hartman, and Arend Rensink. Vol. 5562. Lecture Notes in Computer Science. Springer, 2009, pp. 114–129. ISBN: 978-3-642-02673-7. DOI: 10.1007/978-3-642-02674-4_9.

[58] Florian Heidenreich, Jendrik Johannes, Jan Reimann, Mirko Seifert, Christian Wende, Christian Werner, Claas Wilke, and Uwe Aßmann. "Model-driven Modernisation of Java Programs with JaMoPP". In: *Joint Proceedings of the First International Workshop on Model-Driven Software Migration (MDSM 2011) and the Fifth International Workshop on System Quality and Maintainability (SQM 2011), March 1, 2011 in Oldenburg, Germany*. CEUR Workshop Proceedings, Mar. 2011, pp. 8–11.

[59] Zef Hemel, Danny M. Groenewegen, Lennart C. L. Kats, and Eelco Visser. "Static consistency checking of web applications with WebDSL". In: *Journal of Symbolic Computation* 46.2 (2011), pp. 150–182. DOI: 10.1016/j.jsc.2010.08.006.

[60] Zef Hemel, Lennart C. L. Kats, Danny M. Groenewegen, and Eelco Visser. "Code generation by model transformation: a case study in transformation modularity". In: *Software and Systems Modeling* 9.3 (2010), pp. 375–402. DOI: 10.1007/s10270-009-0136-1.

[61] Zef Hemel and Eelco Visser. "Declaratively programming the mobile web with Mobl". In: *Proceedings of 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, 2011, pp. 695–712. ISBN: 978-1-4503-0940-0. DOI: 10.1145/2048066.2048121.

[62] Jason Hickey and Aleksey Nogin. "OMake: Designing a Scalable Build Process". In: *Fundamental Approaches to Software Engineering, 9th International Conference, FASE 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27-28, 2006, Proceedings*. Ed. by Luciano Baresi and Reiko Heckel. Vol. 3922. Lecture Notes in Computer Science. Springer, 2006, pp. 63–78. ISBN: 3-540-33093-3. DOI: 10.1007/11693017_7.

[63] Sungpack Hong, Hassan Chafi, Eric Sedlar, and Kunle Olukotun. "Green-Marl: a DSL for easy and efficient graph analysis". In: *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2012, London, UK, March 3-7, 2012*. Ed. by Tim Harris and Michael L. Scott. ACM, 2012, pp. 349–362. ISBN: 978-1-4503-0759-8. DOI: 10.1145/2150976.2151013.

[64] B. Hoyts and Simon Alford. *fabricate*. https://github.com/SimonAlfie/fabricate. 2009. (Visited on 10/04/2019).

[65] *Human Usable Textual Notation Specification*. Object Management Group. 2004.

[66] Harry D. Huskey, M. H. Halstead, and R. McArthur. "NELIAC - Dialect of ALGOL". In: *Communications of the ACM* 3.8 (Aug. 1960), pp. 463–468. DOI: 10.1145/367368.367373.

[67] Gradle Inc. *Gradle Build Tool*. https://gradle.org/. (Visited on 10/04/2019).

[68] *interactive*. In: *Cambridge Dictionary*. 2018. URL: https://dictionary.cambridge.org/dictionary/english/interactive (visited on 10/04/2019).

[69] *Java Microbenchmarking Harness (JMH)*. http://openjdk.java.net/projects/code-tools/jmh/. (Visited on 10/04/2019).

[70] *Jenkins Pipeline syntax*. https://jenkins.io/doc/book/pipeline/syntax/. (Visited on 10/04/2019).

[71] JetBrains. *Kotlin Programming Language*. https://kotlinlang.org/. (Visited on 10/04/2019).

[72] Gregory F. Johnson and Charles N. Fischer. "A Meta-Language and System for Nonlocal Incremental Attribute Evaluation in Language-Based Editors". In: *POPL*. 1985, pp. 141–151.

[73] Uwe Kastens. "Ordered Attributed Grammars". In: *Acta Informatica* 13 (1980), pp. 229–256.

[74] Uwe Kastens and William M. Waite. "Modularity and Reusability in Attribute Grammars". In: *Acta Informatica* 31.7 (1994). http://portal.acm.org/citation.cfm?id=191491, pp. 601–627.

[75] Lennart C. L. Kats and Eelco Visser. "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, 2010, pp. 444–463. ISBN: 978-1-4503-0203-6. DOI: 10.1145/1869459.1869497.

[76] Lennart C. L. Kats, Eelco Visser, and Guido Wachsmuth. "Pure and declarative syntax definition: paradise lost and regained". 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, 2010, pp. 918–932. ISBN: 978-1-4503-0203-6. DOI: 10.1145/1869459.1869535.

[77] Sven Keidel, Wulf Pfeiffer, and Sebastian Erdweg. "The IDE portability problem and its solution in Monto". 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. http://dl.acm.org/citation.cfm?id=2997368. ACM, 2016, pp. 152–162. ISBN: 978-1-4503-4447-0.

[78] Steven Kelly, Kalle Lyytinen, and Matti Rossi. "MetaEdit+ A Fully Configurable Multi-User and Multi-Tool CASE and CAME Environment". In: *Seminal Contributions to Information Systems Engineering, 25 Years of CAiSE*. Ed. by Janis A. Bubenko Jr., John Krogstie, Oscar Pastor, Barbara Pernici, Colette Rolland, and Arne Sølvberg. Springer, 2013, pp. 109–129. ISBN: 978-3-642-36926-1. DOI: 10.1007/978-3-642-36926-1_9.

[79] Thilo Kielmann. *PROM: A flexible, PROLOG-based make tool.* Technical Report Report TI-4/91. Darmstadt, Germany: Institute of Theoretical Computer Science, Darmstadt University of Technology, 1991.

[80] Paul Klint. "A Meta-Environment for Generating Programming Environments". In: *ACM Transactions on Software Engineering Methodology* 2.2 (1993), pp. 176–201. DOI: 10.1145/151257.151260.

[81] Paul Klint, Tijs van der Storm, and Jurgen J. Vinju. "RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation". In: *Ninth IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2009, Edmonton, Alberta, Canada, September 20-21, 2009*. IEEE Computer Society, 2009, pp. 168–177. ISBN: 978-0-7695-3793-1. DOI: 10.1109/SCAM.2009.28.

[82] Donald E. Knuth. "backus normal form vs. Backus Naur form". In: *Communications of the ACM* 7.12 (1964), pp. 735–736. DOI: 10.1145/355588.365140.

[83] Donald E. Knuth. "Semantics of Context-Free Languages". In: *Theory Comput. Syst.* 2.2 (1968). http://www.springerlink.com/content/m2501m07m4666813/, pp. 127–145.

[84] Gabriel Konat. *PIE implementation for <Programming> 2018*. Mar. 2018. DOI: 10.5281/zenodo.1199192.

[85] Gabriel Konat. *Spoofax-PIE implementation for <Programming> 2018*. Mar. 2018. DOI: 10.5281/zenodo.1199194.

[86] Gabriël Konat, Sebastian Erdweg, and Eelco Visser. "Bootstrapping Domain-Specific Meta-Languages in Language Workbenches". In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2016, Amsterdam, The Netherlands, October 31 - November 1, 2016*. Ed. by Bernd Fischer and Ina Schaefer. ACM, 2016, pp. 47–58. ISBN: 978-1-4503-4446-3. DOI: 10.1145/2993236.2993242.

[87] Gabriël Konat, Sebastian Erdweg, and Eelco Visser. "Scalable incremental building with dynamic task dependencies". In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*. Ed. by Marianne Huchard, Christian Kästner, and Gordon Fraser. ACM, 2018, pp. 76–86. DOI: 10.1145/3238147.3238196.

[88] Gabriël Konat, Sebastian Erdweg, and Eelco Visser. "Towards Live Language Development". In: *Workshop on Live Programming Systems (LIVE)*. 2016.

[89] Gabriël Konat, Lennart C. L. Kats, Guido Wachsmuth, and Eelco Visser. "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, 2012, pp. 311–331. ISBN: 978-3-642-36089-3. DOI: 10.1007/978-3-642-36089-3_18.

[90] Gabriël Konat, Luís Eduardo de Souza Amorim, Sebastian Erdweg, and Eelco Visser. *Bootstrapping, Default Formatting, and Skeleton Editing in the Spoofax Language Workbench*. Language Workbench Challenge (LWC@SLE). https://2016.splashcon.org/details/lwc2016/4/Bootstrapping-Default-Formatting-and-Skeleton-Editing-in-the-Spoofax-Language-Workb. 2016.

[91] Gabriël Konat, Michael J. Steindorfer, Sebastian Erdweg, and Eelco Visser. "PIE: A Domain-Specific Language for Interactive Software Development Pipelines". In: *Programming Journal* 2.3 (2018), p. 9. DOI: 10.22152/programming-journal.org/2018/2/9.

[92] Shriram Krishnamurthi. *Programming Languages: Application and Interpretation*. http://www.cs.brown.edu/~sk/Publications/Books/ProgLangs/2007-04-26/. 2007.

[93] Epperly T Kumfert G. *Software in the DOE: The Hidden Overhead of "The Build"*. Tech. rep. Lawrence Livermore National Laboratory, 2002.

[94] Oracle Labs. *Parallel Graph AnalytiX*. https://www.oracle.com/technetwork/oracle-labs/parallel-graph-analytix/overview/index.html. (Visited on 10/04/2019).

[95] Olivier Lecarme, Mireille Pellissier, and Marie-Claude Thomas. "Computer-aided Production of Language Implementation Systems: A Review and Classification". In: *Software: Practice and Experience* 12.9 (1982), pp. 785–824.

[96] John Levine. *Flex & Bison*. O'Reilly Series. O'Reilly Media, 2009. ISBN: 9780596155971.

[97] Alex Loh, Tijs van der Storm, and William R. Cook. "Managed data: modular strategies for data abstraction". In: *ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2012, part of SPLASH '12, Tucson, AZ, USA, October 21-26, 2012*. Ed. by Gary T. Leavens and Jonathan Edwards. ACM, 2012, pp. 179–194. ISBN: 978-1-4503-1562-3. DOI: 10.1145/2384592.2384609.

[98] David Mackenzie, Tom Tromey, Alexandre Duret-Lutz, Ralf Wildenhues, and Stefano Lattarini. *GNU Automake*. Free Software Foundation, Feb. 2018.

[99] Evan Martin. *The Ninja build system*. https://ninja-build.org/manual.html. (Visited on 10/04/2019).

[100] Shane McIntosh, Bram Adams, and Ahmed E. Hassan. "The evolution of ANT build systems". In: *Proceedings of the 7th International Working Conference on Mining Software Repositories, MSR 2010 (Co-located with ICSE), Cape Town, South Africa, May 2-3, 2010, Proceedings*. Ed. by Jim Whitehead and Thomas Zimmermann. IEEE, 2010, pp. 42–51. ISBN: 978-1-4244-6803-4. DOI: 10.1109/MSR.2010.5463341.

[101] Erik Meijer. "Reactive extensions (Rx): curing your asynchronous programming blues". In: *ACM SIGPLAN Commercial Users of Functional Programming*. ACM. 2010, p. 11. DOI: 10.1145/1900160.1900173.

[102] Marjan Mernik, Jan Heering, and Anthony M. Sloane. "When and how to develop domain-specific languages". In: *ACM Computing Surveys* 37.4 (2005), pp. 316–344. DOI: 10.1145/1118890.1118892.

[103] Emma Anna Van Der Meulen. "Incremental Rewriting". PhD thesis. University of Amsterdam, 1994.

[104] Jon Meyer and Troy Downing. *Java Virtual Machine*. O Reilly, 1997. ISBN: 1-56592-194-1.

[105] Heather Miller, Philipp Haller, and Martin Odersky. "Spores: A Type-Based Foundation for Closures in the Age of Concurrency and Distribution". In: *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings*. Ed. by Richard Jones. Vol. 8586. Lecture Notes in Computer Science. Springer, 2014, pp. 308–333. ISBN: 978-3-662-44201-2. DOI: 10.1007/978-3-662-44202-9_13.

[106] Peter Miller. *Recursive make considered harmful*. `http://aegis.sourceforge.net/auug97.pdf`. (Visited on 10/04/2019).

[107] Neil Mitchell. "Shake before building: replacing make with haskell". In: *ACM SIGPLAN International Conference on Functional Programming, ICFP'12, Copenhagen, Denmark, September 9-15, 2012*. Ed. by Peter Thiemann and Robby Bruce Findler. ACM, 2012, pp. 55–66. ISBN: 978-1-4503-1054-3. DOI: 10.1145/2364527.2364538.

[108] Andrey Mokhov, Neil Mitchell, Simon L. Peyton Jones, and Simon Marlow. "Non-recursive make considered harmful: build systems at scale". In: *Proceedings of the 9th International Symposium on Haskell, Haskell 2016, Nara, Japan, September 22-23, 2016*. Ed. by Geoffrey Mainland. ACM, 2016, pp. 170–181. ISBN: 978-1-4503-4434-0. DOI: 10.1145/2976002.2976011.

[109] Pierre Néron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. "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, 2015, pp. 205–231. ISBN: 978-3-662-46668-1. DOI: 10.1007/978-3-662-46669-8_9.

[110] K. V. Nori, U. Ammann, K. Jensen, and H. H. Nägeli. *The PASCAL <P> Compiler: Implementation Notes*. Tech. rep. ETH Zürich, 1974.

[111] *Object Constraint Language*. 2.3.1. Object Management Group. 2012.

[112] Martin Odersky. "Defining Context-Dependent Syntax Without Using Contexts". In: *ACM Transactions on Programming Languages and Systems* 15.3 (1993), pp. 535–562. DOI: 10.1145/169683.174159.

[113] Hubert Österle, Jörg Becker, Ulrich Frank, Thomas Hess, Dimitris Karagiannis, Helmut Krcmar, Peter Loos, Peter Mertens, Andreas Oberweis, and Elmar J. Sinz. "Memorandum on design-oriented information systems research". In: *EJIS* 20.1 (2011), pp. 7–10. DOI: 10.1057/ejis.2010.55.

[114] Daniël Pelsmaeker. "Portable Editor Services". MA thesis. 2018.

[115] Maarten C. Pennings. "Generating incremental attribute evaluators". Ph.D. Thesis. Computer Science, Utrecht University, Nov. 1994.

[116] *PIE implementation*. https://github.com/metaborg/pie. (Visited on 10/04/2019).

[117] Benjamin C. Pierce. *Types and Programming Languages*. Cambridge, Massachusetts: MIT Press, 2002.

[118] Arnd Poetzsch-Heffter. "Implementing High-Level Identification Specifications". In: *Compiler Construction, 4th International Conference on Compiler Construction, CC 92, Paderborn, Germany, October 5-7, 1992, Proceedings*. Ed. by Uwe Kastens and Peter Pfahler. Vol. 641. Lecture Notes in Computer Science. Springer, 1992, pp. 59–65. ISBN: 3-540-55984-1.

[119] Arnd Poetzsch-Heffter. "Logic-Based Specification of Visibility Rules". In: *PLILP*. 1991, pp. 63–74.

[120] Arnd Poetzsch-Heffter. "Programming Language Specification and Prototyping Using the MAX System". In: *Programming Language Implementation and Logic Programming, 5th International Symposium, PLILP 93, Tallinn, Estonia, August 25-27, 1993, Proceedings*. Ed. by Maurice Bruynooghe and Jaan Penjam. Vol. 714. Lecture Notes in Computer Science. Springer, 1993, pp. 137–150. ISBN: 3-540-57186-8.

[121] Thomas W. Reps. *Generating language-based environments*. Cambridge, MA, USA: Massachusetts Institute of Technology, 1984. ISBN: 0-262-18115-0.

[122] Thomas W. Reps. "Optimal-Time Incremental Semantic Analysis for Syntax-Directed Editors". In: *POPL*. 1982, pp. 169–176. DOI: 10.1145/582153.582172.

[123] Thomas W. Reps and Tim Teitelbaum. "The Synthesizer Generator". In: *Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*. New York, USA: ACM, 1984, pp. 42–48. DOI: 10.1145/800020.808247.

[124] Thomas W. Reps, Tim Teitelbaum, and Alan J. Demers. "Incremental Context-Dependent Analysis for Language-Based Editors". In: *ACM Transactions on Programming Languages and Systems* 5.3 (1983), pp. 449–477. DOI: 10.1145/2166.357218.

[125] Tiark Rompf and Martin Odersky. "Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs". In: *Generative Programming And Component Engineering, Proceedings of the Ninth International Conference on Generative Programming and Component Engineering, GPCE 2010, Eindhoven, The Netherlands, October 10-13, 2010*. Ed. by Eelco Visser and Jaakko Järvi. ACM, 2010, pp. 127–136. ISBN: 978-1-4503-0154-1. DOI: 10.1145/1868294.1868314.

[126] Guido Salvaneschi, Gerold Hintz, and Mira Mezini. "REScala: bridging between object-oriented and functional style in reactive applications". In: *13th International Conference on Modularity, MODULARITY '14, Lugano, Switzerland, April 22-26, 2014*. Ed. by Walter Binder, Erik Ernst, Achille Peternier, and Robert Hirschfeld. ACM, 2014, pp. 25–36. ISBN: 978-1-4503-2772-5. DOI: 10.1145/2577080.2577083.

[127] Mike Shal. *Build System Rules and Algorithms*. http://gittup.org/tup/build_system_rules_and_algorithms.pdf. 2009. (Visited on 10/04/2019).

[128] Mary Shaw. "Writing Good Software Engineering Research Papers". In: *Proceedings of the 25th International Conference on Software Engineering, May 3-10, 2003, Portland, Oregon, USA*. http://computer.org/proceedings/icse/1877/18770726abs.htm. IEEE Computer Society, 2003, pp. 726–737.

[129] Jeff Smits and Eelco Visser. "FlowSpec: declarative dataflow analysis specification". 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, 2017, pp. 221–231. ISBN: 978-1-4503-5525-4. DOI: 10.1145/3136014.3136029.

[130] Emma Söderberg. "Contributions to the Construction of Extensible Semantic Editors". PhD thesis. 2012.

[131] Emma Söderberg and Görel Hedin. "A Comparative Study of Incremental Attribute Grammar Solutions to Name Resolution". In: 2012.

[132] *Spoofax-PIE implementation*. https://github.com/metaborg/spoofax-pie. (Visited on 10/04/2019).

[133] Richard M. Stallman, Roland McGrath, and Paul D. Smith. *GNU Make*. Free Software Foundation, May 2016.

[134] *Standard ECMA-334 C# Language Specification, 5th edition*. 2017.

[135] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *Eclipse Modeling Framework*. 2nd ed. Addison-Wesley, 2009.

[136] Michael J. Steindorfer and Jurgen J. Vinju. *Artifact for 'Optimizing Hash-Array Mapped Tries for Fast and Lean Immutable JVM Collections'*. https://github.com/msteindorfer/oopsla15-artifact. 2015. (Visited on 10/04/2019).

[137] Michael J. Steindorfer and Jurgen J. Vinju. "Optimizing hash-array mapped tries for fast and lean immutable JVM collections". In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*. Ed. by Jonathan Aldrich and Patrick Eugster. ACM, 2015, pp. 783–800. ISBN: 978-1-4503-3689-5. DOI: 10.1145/2814270.2814312.

[138] Tijs van der Storm, William R. Cook, and Alex Loh. "Object Grammars". 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, 2012, pp. 4–23. ISBN: 978-3-642-36089-3. DOI: 10.1007/978-3-642-36089-3_2.

[139] S. Doaitse Swierstra and Harald Vogt. "Higher Order Attribute Grammars". 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, 1991, pp. 256–296. ISBN: 3-540-54572-7.

[140] Symas. *Lightning Memory-mapped Database*. https://symas.com/lmdb/. (Visited on 10/04/2019).

[141] Tamás Szabó, Sebastian Erdweg, and Markus Völter. "IncA: a DSL for the definition of incremental program analyses". In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*. Ed. by David Lo, Sven Apel, and Sarfraz Khurshid. ACM, 2016, pp. 320–331. ISBN: 978-1-4503-3845-5. DOI: 10.1145/2970276.2970298.

[142] M. Levin T. Hart. *AI Memo 39 - The New Compiler*. Tech. rep. MIT, 1962.

[143] Walid Taha and Tim Sheard. "MetaML and multi-stage programming with explicit annotations". In: *Theoretical Computer Science* 248.1-2 (2000), pp. 211–242. DOI: 10.1016/S0304-3975(00)00053-0.

[144] *Task Engine Benchmarking Results*. https://bitbucket.org/slde/opendata-experiments. (Visited on 10/04/2019).

[145] Delft University of Technology. *TUDelft Compiler Construction, Lab 6: Name Analysis*. http://tudelft-in4303.github.io/assignments/ms2/lab6.html. (Visited on 10/04/2019).

[146] Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. "Languages as libraries". In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*. Ed. by Mary W. Hall and David A. Padua. ACM, 2011, pp. 132–141. ISBN: 978-1-4503-0663-8. DOI: 10.1145/1993498.1993514.

[147] Vlad A. Vergu, Pierre Néron, and Eelco Visser. "DynSem: A DSL for Dynamic Semantics Specification". In: *26th International Conference on Rewriting Techniques and Applications, RTA 2015, June 29 to July 1, 2015, Warsaw, Poland*. Ed. by Maribel Fernández. Vol. 36. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015, pp. 365–378. ISBN: 978-3-939897-85-9. DOI: 10.4230/LIPIcs.RTA.2015.365.

[148] Sander Vermolen, Chris Melman, Elmer van Chastelet, Danny Groenewegen, and Eelco Visser. *YellowGrass source code*. `https://github.com/webdsl/yellowgrass`. (Visited on 10/04/2019).

[149] Sander Vermolen, Chris Melman, Elmer van Chastelet, Danny Groenewegen, and Eelco Visser. *YellowGrass: a tag-based issue tracker powered by WebDSL*. `https://yellowgrass.org/`. (Visited on 10/04/2019).

[150] Eelco Visser. "A Bootstrapped Compiler for Strategies (Extended Abstract)". In: *Strategies in Automated Deduction (STRATEGIES'99)*. Trento, Italy, July 1999, pp. 73–83.

[151] Eelco Visser. "Program Transformation with Stratego/XT: Rules, Strategies, Tools, and Systems in Stratego/XT 0.9". In: *Domain-Specific Program Generation, International Seminar, Dagstuhl Castle, Germany, March 23-28, 2003, Revised Papers*. Ed. by Christian Lengauer, Don S. Batory, Charles Consel, and Martin Odersky. Vol. 3016. Lecture Notes in Computer Science. Springer, 2003, pp. 216–238. ISBN: 3-540-22119-0. DOI: 10.1007/978-3-540-25935-0_13.

[152] Eelco Visser. *Scannerless Generalized-LR Parsing*. Tech. rep. P9707. Programming Research Group, University of Amsterdam, July 1997.

[153] Eelco Visser. "Syntax Definition for Language Prototyping". PhD thesis. University of Amsterdam, Sept. 1997.

[154] Eelco Visser. *WebDSL Blog source code*. `https://github.com/webdsl/blog`. (Visited on 10/04/2019).

[155] Eelco Visser, Guido Wachsmuth, Andrew P. Tolmach, Pierre Néron, Vlad A. Vergu, Augusto Passalaqua, and Gabriël Konat. "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, 2014, pp. 95–111. ISBN: 978-1-4503-3210-1. DOI: 10.1145/2661136.2661149.

[156] Harald Vogt, S. Doaitse Swierstra, and Matthijs F. Kuiper. "Efficient Incremental Evaluation of Higher order Attribute Grammars". In: *PLILP*. 1991, pp. 231–242. DOI: 10.1007/3-540-54444-5_102.

[157] Harald Vogt, S. Doaitse Swierstra, and Matthijs F. Kuiper. "Higher-Order Attribute Grammars". In: *PLDI*. 1989, pp. 131–145.

[158] Tobi Vollebregt, Lennart C. L. Kats, and Eelco Visser. "Declarative specification of template-based textual editors". In: *International Workshop on Language Descriptions, Tools, and Applications, LDTA '12, Tallinn, Estonia, March 31 - April 1, 2012*. Ed. by Anthony Sloane and Suzana Andova. ACM, 2012, pp. 1–7. ISBN: 978-1-4503-1536-4. DOI: 10.1145/2427048.2427056.

[159] Markus Völter and Konstantin Solomatov. "Language Modularization and Composition with Projectional Language Workbenches illustrated with MPS". In: *Software Language Engineering, Third International Conference, SLE 2010*. Ed. by Mark G. J. van den Brand, Brian Malloy, and Steffen Staab. Lecture Notes in Computer Science. Springer, 2010.

[160] Guido Wachsmuth, Gabriël Konat, Vlad A. Vergu, Danny M. Groenewegen, and Eelco Visser. "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, 2013, pp. 260–280. ISBN: 978-3-319-02653-4. DOI: 10.1007/978-3-319-02654-1_15.

[161] Guido Wachsmuth, Gabriël Konat, and Eelco Visser. "Language Design with the Spoofax Language Workbench". In: *IEEE Software* 31.5 (2014), pp. 35–43. DOI: 10.1109/MS.2014.100.

[162] Edwin Westbrook, Mathias Ricken, Jun Inoue, Yilong Yao, Tamer Abdelatif, and Walid Taha. "Mint: Java multi-stage programming using weak separability". In: *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*. Ed. by Benjamin G. Zorn and Alexander Aiken. ACM, 2010, pp. 400–411. ISBN: 978-1-4503-0019-3. DOI: 10.1145/1806596.1806642.

[163] Eric Van Wyk, Derek Bodin, Jimin Gao, and Lijesh Krishnan. "Silver: An extensible attribute grammar system". In: *Science of Computer Programming* 75.1-2 (2010), pp. 39–54. DOI: 10.1016/j.scico.2009.07.004.

[164] Dashing Yeh. "On Incremental Evaluation of Ordered Attribute Grammars". In: *BIT* 23.3 (1983), pp. 308–320. DOI: 10.1007/BF01934460.

[165] Dashing Yeh and Uwe Kastens. "Improvements of an incremental evaluation algorithm for ordered attribute grammars". In: *SIGPLAN Notices* 23.12 (1988), pp. 45–50. DOI: 10.1145/57669.57672.

Curriculum Vitae

Gabriël Ditmar Primo Konat

Born August 13th 1988 in The Hague, the Netherlands.

2018 - present
Postdoctoral Research
Delft University of Technology
Department of Software Technology, Programming Languages group

2012 - 2018
Ph.D. in Computer Science
Delft University of Technology
Department of Software Technology, Programming Languages group

06/2013 - 08/2013, 07/2014 - 09/2014, 07/2015 - 09/2015
Research Assistant
Oracle Labs in Redwood Shores, California, United States of America

2009 - 2012
M.Sc. in Computer Science (cum laude)
Delft University of Technology
Specialization: Software Engineering

2005 - 2009
B.Sc. in Computer Science
Institute of Applied Sciences, Rijswijk
Specialization: Software Development

2000 - 2005
HAVO diploma
Segbroek College in The Hague
Specialization: Nature & Technology

List of Publications

- Gabriël Konat, Sebastian Erdweg, and Eelco Visser. "Scalable incremental building with dynamic task dependencies". In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*. Ed. by Marianne Huchard, Christian Kästner, and Gordon Fraser. ACM, 2018, pp. 76–86. DOI: 10.1145/3238147.3238196
- Gabriël Konat, Michael J. Steindorfer, Sebastian Erdweg, and Eelco Visser. "PIE: A Domain-Specific Language for Interactive Software Development Pipelines". In: *Programming Journal* 2.3 (2018), p. 9. DOI: 10.22152/programming-journal.org/2018/2/9
- Gabriël Konat, Sebastian Erdweg, and Eelco Visser. "Bootstrapping Domain-Specific Meta-Languages in Language Workbenches". In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2016, Amsterdam, The Netherlands, October 31 - November 1, 2016*. Ed. by Bernd Fischer and Ina Schaefer. ACM, 2016, pp. 47–58. ISBN: 978-1-4503-4446-3. DOI: 10.1145/2993236.2993242
- Gabriël Konat, Luís Eduardo de Souza Amorim, Sebastian Erdweg, and Eelco Visser. *Bootstrapping, Default Formatting, and Skeleton Editing in the Spoofax Language Workbench*. Language Workbench Challenge (LWC@SLE). https://2016.splashcon.org/details/lwc2016/4/Bootstrapping-Default-Formatting-and-Skeleton-Editing-in-the-Spoofax-Language-Workb. 2016
- Gabriël Konat, Sebastian Erdweg, and Eelco Visser. "Towards Live Language Development". In: *Workshop on Live Programming Systems (LIVE)*. 2016
- Sebastian Erdweg, 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. "Evaluating and comparing language workbenches: Existing results and benchmarks for the future". In: *Computer Languages, Systems & Structures* 44 (2015), pp. 24–47. DOI: 10.1016/j.cl.2015.08.007
- Eelco Visser, Guido Wachsmuth, Andrew P. Tolmach, Pierre Néron, Vlad A. Vergu, Augusto Passalaqua, and Gabriël Konat. "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, 2014, pp. 95–111. ISBN: 978-1-4503-3210-1. DOI: 10.1145/2661136.2661149

- Guido Wachsmuth, Gabriël Konat, and Eelco Visser. "Language Design with the Spoofax Language Workbench". In: *IEEE Software* 31.5 (2014), pp. 35–43. DOI: 10.1109/MS.2014.100
- Sebastian Erdweg, 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. "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, 2013, pp. 197–217. ISBN: 978-3-319-02653-4. DOI: 10.1007/978-3-319-02654-1_11
- Guido Wachsmuth, Gabriël Konat, Vlad A. Vergu, Danny M. Groenewegen, and Eelco Visser. "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, 2013, pp. 260–280. ISBN: 978-3-319-02653-4. DOI: 10.1007/978-3-319-02654-1_15
- Gabriël Konat, Lennart C. L. Kats, Guido Wachsmuth, and Eelco Visser. "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, 2012, pp. 311–331. ISBN: 978-3-642-36089-3. DOI: 10.1007/978-3-642-36089-3_18