Strategic Language Workbench Improvements

Jeff Smits

Strategic Language Workbench Improvements

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
Friday 8 September 2023 at 10:00 o'clock

by

Jeff SMITS
Master of Science in Computer Science
Delft University of Technology, the Netherlands
born in Moordrecht, the Netherlands

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

The work in this dissertation has been carried out at the Delft University of Technology under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

This research was supported by the NWO VICI Language Designer's Workbench project (639.023.206) and unrestricted gifts from the Oracle Corporation.

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

ISBN: 978-94-6384-474-1

time is a cruel mistress
you seemingly have plenty
but some run out so suddenly

you need to make choices
what to spend your time on
too many options, interests

my choices led to this
I am… content
with those choices

now let me ask you
dear reader
are you?

Preface

So how did I end up writing this dissertation? Well, you could say there were some signs that this was the direction I would take. Allow me to reminisce here for a bit, before I tell you something about the book 'design' and then acknowledge the many people who've been there for me over the years.

The Road to Nerding Out over Programming Languages

My parents like to tell the story of how my first exposure to computers was at a very young age. My sister—two years my senior—excitedly showed me how press a button on the computer to make Donald Duck quack. I don't remember this, I was too young, but I can recall the old computer in the living room, a thick CRT screen with yellow tinged white plastic shell on a white, formica desk.

My father was an SAP consultant for a good part of my youth, so we had a nice enough computer in the living room, and dial-up internet, used sparingly because of the cost. I grew up with an interest in many things, and although I was quite computer literate[1], I did not show much interest in programming. This was all fine, my father is not the type to pressure his children into the same career.

Secondary School. At some point in my mid teens, I got to do an extra course in school which was presented to us as *Programming in Java*. I'm afraid I did not learn any Java in that course, there was very little material that we got taught in classroom lessons. But we did have a number of computers with internet (ADSL by now), and so I mostly remember us kids figuring out how to make little websites with JavaScript copied and pasted from websites with collections of snippets.

With my interest already raised, my dad now offered to teach me a bit of programming. He dug up a CD of Borland C++ from the big stack, and grabbed his copy of Kernighan & Ritchie's *The C Programming Language*. I remember asking what *studio dot h* was, and learning that `stdio.h` was the library for *standard input/output*. And that I didn't get much further than a temperature conversion command line program[2], before going back to messing with websites, since those had a faster, more visual feedback loop.

Nearing the end of secondary school, I was taking art classes as an elective. Originally I had just wanted to do something with my hands instead of using only my brain and writing skills, and it basically remained interesting and fun so I stuck with it. In particular, I liked architecture, and together with my

[1] I could use office suite programs, an internet browser, and an image editor.

[2] You know, give a temperature in Celsius, and it will print what the temperature is in ~~antiquated units~~ Fahrenheit.

interest in physics, I had figured out what I wanted to do. Architecture would combine my interests in aesthetics, in physics and material properties, and to some extent in the social function and liveability of buildings.

Programming websites was just a secondary hobby, where I was starting to scratch an itch: the CSS language was underpowered and I was trying to create an extension with variables and other code sharing features[3].

The nearest university with an architecture programme was TU Delft, a university that I figured had some international standing, since their student teams were in the news about solar car racing in a competition abroad[4]. So I went to the university's open day and I filled in my three options to take a look at: architecture, and the unnecessary backup options of computer science and physics. I was *very* disappointed to learn that architecture was a study programme strictly separate from civil engineering. My impression was that I could learn about aesthetics and liveability at architecture, and learn about physically achievable building design at civil engineering though they seemed to prefer larger infrastructure design. There was also no mention of support for a double bachelor, and besides, I wasn't that good of a student, so I would probably not have considered such a heavy programme.

Somewhat deflated, or maybe more frustrated (I was a teenager after all), I continued to see about computer science and physics, and computer science looked pretty neat. I was also lucky enough to have a friend one year ahead of me in school, who had gone to Delft to do computer science, so I learned a little about the programme through him.

Bachelor. And so I ended up in Delft, studying computer science. I found myself so thoroughly enjoying the courses, that I knew I had made the right choice pretty quickly and stopped feeling sore about architecture. Finding myself scoring high grades again, I re-found the joy of learning interesting things and getting good scores on tests. I also learned that as a student, you could become a teaching assistant for a course the year after you had successfully completed it. It looked easy, fun, and paid much better than my previous part-time job stocking shelves at a grocery store.

Through that job as a teaching assistant is how I first got to know Eelco Visser a little. He had taught *Concepts of Programming Languages* for the first time in the year that I started my bachelor programme, and asked me to be a TA for the subsequent year. I ended up being a TA for that course for five years straight.

Eelco was a charismatic teacher with a real enthusiasm for the material he taught. He pushed students to work hard and learn a lot, and as someone with an interest in what he taught, I loved it. Which isn't to say he didn't have his flaws, my usual gripe as a student applied to him as it did to most teachers: I thought professors had less time and interest for teaching than for research.

[3]Since I did not know and had not found any material on compilers, I was doing some text processing with regular expressions, and did not get very far before I learned about a project named SASS that was doing the same I wanted to do.

[4]Yes, really, I based my impression of the international standing of a local university on national news coverage; not an entirely sound argument eh?

But as a TA I already learned that Eelco would work on everything until the last minute, although it didn't sink in until later just how ridiculously busy he always was.

I doubt it's surprising that I ended up doing my bachelor's thesis project with him. Through that project, I was first exposed to the Spoofax language workbench. By the end of the thesis project, I told Eelco I wanted to do a PhD. I had this romantic image of what doing research would be like... Eelco told me it was a bit early to speak of PhDs. I could first do a master's thesis with his group, as long as I took their courses during my master. But if I still wanted to do a PhD by the end of my master's, he would make sure that he had some funding for a position.

Master. I was hooked. I had already planned out my master's programme at TU Delft anyway, and that did already include the courses by Eelco and his colleagues in the subgroup of *Software Language Design Engineering*[5]. By naively assuming the master courses would take a similar amount of time as the bachelor courses, I figured I could continue to take the courses that were said to be hard (because those were the interesting ones to me) in combination with some other things. In particular, I remained a TA for multiple courses, and on top of it, I would try that Honours Programme thing that I had already been invited to once during my bachelor.

In my hubris, I had thought I could just learn my limits this way by taking on more things simultaneously. If it really turned out to be too much, I could just tough it out and drop some things I had planned for later in the year, right? What I hadn't taken into consideration was that courses at the master level were actually significantly more work, and that I did *not* like giving up on anything. Of course the alternative was not exactly nice for others: I gave a bit of a bad impression by asking Eelco what the minimum requirements were to finish my honours programme project with him (yes, of course I went back to him). I also, for the first time in my studies, did significantly less work than another in a group project for a course. This was particularly embarrassing because my partner in that project had picked me because I had a good reputation for group work.

Despite these struggles, I managed to stay on schedule and as I got closer to the time to pick a master's thesis project, I was not too sure what I wanted. So Eelco suggested a topic he had ready, for which I'd also go on an internship in Silicon Valley. I remember well how I first agreed to do that, and only then asked more details about the topic. It turned out that Eelco had good connections with a research group within Oracle Labs, the research department of Oracle. They had a compiler written in C++ for a domain specific language called GreenMarl. A previous attempt at porting this compiler over to Spoofax had failed, but they had a grammar available from that. I wasn't surprised they wanted to use Spoofax, I still had a minor vendetta against C++ after my experience using it during my robotics minor during my BSc. I went to

[5]I had originally considered other focusses like computer graphics or intelligent systems, but computer graphics seemed to require plenty of statistics, which I found a difficult, unintuitive part of mathematics. For intelligent systems I hadn't taken the right electives during my BSc.

stay in California for 6 months to work on this GreenMarl compiler. My thesis advisor in Delft was Guido Wachsmuth, who had also taught the Compiler Construction course. He advised me to start writing early, which I earnestly attempted a few times, but got stuck with fairly quickly. I ended up back in Delft with a working GreenMarl compiler in Spoofax and very little on paper. And while writing up what I'd done, I found that my intuitive understanding of the static semantics of GreenMarl was challenging to write up. So I ended up taking six months to write up everything, including a somewhat readable formalisation of the static semantics of GreenMarl (the first ever published). But I must have driven Guido a little crazy because I was not a good author at the time. Nevertheless, he helped me through it, and after my defence he advised me not to do a PhD because it would be the same writing struggle all over again many times. Fair advice, but I wanted to follow through with my plan that I had made long ago with much less information. I don't like giving up on things, remember? Silly, stubborn me.

PhD. With my troubles in writing and formalising things, Eelco thought it a bad idea to put me on his Vici project which would very likely involve all kinds of formalisations. I disagreed, but when Arjen Rouvoet later got that position, he certainly did a stellar job. But Eelco kept his word, he had a position for me from some left-over funds here, and some other money there, which he'd gotten from the Extension School of TU Delft. He'd promised the Extension School to have some teaching related research done with that money. I was not convinced by the topic, which we discussed, but Eelco managed to persuade me to try anyway. He said we'd see how I got along, and if I still didn't like it after a year, we could reconsider.

I lasted ten months, then I'd had enough. I didn't like the topic, I missed direction and supervision, and performing poorly had hurt my confidence. My relationship with Eelco had suffered too, after a particular discussion we'd had where he'd hurt my confidence even more. I still regret not clearing the air about, I was being too conflict-avoiding, and now it's too late...
Funny how I thought I'd simply put it behind me on my own, but now I think back and wish I would have brought it up, even if it would have been years later.

Nevertheless, after ten months, I declaimed that I needed a different topic. I had three ideas of things I would be enthusiastic enough about to be more self-motivating. One of them was to work on improving Stratego, but Eelco preferred to work on that himself since he had some ideas. Another was a new meta-language for control- and data-flow analysis. I don't remember the third, and it's not really relevant here. We went with control- and data-flow analysis, and that's how I ended up truly starting on the work that you can read here, in this dissertation.

The Book You're Reading

Typesetting. I've been working on making my research presentable in this book for a good while. Moving papers that may be in double-column style to a single column, smaller page, different margins document is painful. Especially getting text and figures to end up somewhat close to each other when the figures take up much more space than the text that comes with it. I also micro-managed details such as eliminating final sentences of paragraphs that go past a page turn (although I can't promise I managed with all of them).

Typesetting is something you learn as a computer scientist with the LaTeX tooling, which is used in academic publishing. So I typeset things myself, starting from the typesetting code of Gabriël Konat's dissertation, which got me started quickly. I'm passing on my typesetting code to the next PhD student in our group, where it will probably grow further.

Cover. The cover design is based on a couple of ideas. I personally really like the style period of Art Nouveau (a.k.a. Jugendstil or Modernisma), so for the cover I took inspiration from the poster art of that period. I picked a freely available derivative of the iconic Arnold Böcklin font for the lettering, and reshaped the title to fit the surrounding image. Some of the background elements are merely things from nature, as are often featured in Art Nouveau, although I did pick two things to connect to the work: The tree refers to the tree-shaped data that Stratego does transformations on, while the river is a *flow* of water with some fish in it as FlowSpec looks at the *flow* of control and data following it. Leaves come back in the typesetting as little ornaments here and there. It might have been easier and prettier to get some professional help with the cover, but I've had fun connecting with my much neglected artistic side again.

Dedication. At first, I put an ironic *For Science!* in as (placeholder) dedication, which I thought some of my meme-savvy friends might appreciate as a little joke. Then I had some inspiration for free-form poetry, so I jotted that down. Looking at it now… it's a little strange, a little heavy. But I know the thoughts and emotions behind the words, so I decided to leave it in.

Acknowledgements

First and foremost, I must acknowledge my late promoter, supervisor and mentor for many years: Eelco Visser. As I wrote earlier in this preface, Eelco was a near constant influence on my student life, and has taught me many things, both directly—in courses and conversation—and indirectly. He has taught me all about Program Language research, and about academic writing. I wish Eelco was still with us, but I am thankful for the time I've had to get to know and learn from him.

I thank my current advisors, Arie van Deursen and Jesper Cockx, for taking over and helping me finish this dissertation. Arie was of great help in advising me how to organise the introduction and conclusion of a dissertation, and both Arie and Jesper have given me tips on how to improve my writing there.

I also thank the independent members of the committee: Eric Van Wyk, Ralf Lämmel, Tijs van der Storm, Mathijs de Weerdt and Geert-Jan Houben. It is not an easy task to receive a book like this, and, within a few weeks, determine its merit.

I should not forget to thank the anonymous reviewers of SLE, COMLAN, Programming for their reviews with valuable comments and suggestions of my (draft) papers that I submitted over the years. For the paper from which Chapter 4 is derived, I also want to thank Michael Greenberg for his comments.

I want to thank my master's thesis advisor, Guido Wachsmuth, for all of his advice during my master's thesis, and his guidance on my later internships at Oracle Labs during my PhD. I've tried to copy Guido's helpful, relaxed, and yet enthusiastic style of advising students when I advised master students during my PhD.

These explicit 'thank you's are getting old pretty quickly, so from here on I'm going to write more about the people in my life I appreciate without the explicit words of thanks. I'm sure you can fill them in yourself.

The other (former) professors of the PL group made/make so much of the work in the group possible, including my own. In particular, Sebastian Erdweg gave me great advice at just the right time, which resulted in the publication of the work in Chapter 5; Robbert Krebbers taught me how to use proof assistants and gave me a better understanding of the underlying theory of type systems; Casper Bach Poulsen has been a welcome discussion partner on my research interests and helping to keep Spoofax alive; And Peter Mosses has been a great help in putting research into context with all the work that he has read about and still keeps organised in his head. Peter suggested a particular formulation for semantics on the control-flow rules that I used in Chapter 2.

A research group full of academics is wonderful to be a part of, but for a group to function within the university, you need someone who *gets things done*. That would be the group secretary, which for years was Roniet Sharabi. Roniet would check in with the PhD students once in a while, make sure we were taking breaks and weren't becoming too stressed. Roniet really extended the general welcoming feeling of the PL group.
These days we have Shelley Dawn Stok as secretary after being without a secretary for a while. It is a relief to have a good secretary again, and Shelley certainly fits that role, thinking of our needs even before we do sometimes.

My fellow PhD students and postdocs of the group have been a great support. This support included, but was not limited to, discussing (ongoing) research, proof-reading paper drafts, co-authoring papers, discussing related work, commiserating about difficult supervisors, discussing politics, life, or other idle chitchat. I made a list of names of all the PhD students and postdocs I've gotten to know through our research group, but after getting to 21 names, I figured it would be a tedious list of names to read. If you were a PhD student, postdoc, or something adjacent in the PL group in Delft and met me then, know that I've thought of you explicitly while writing this, and I appreciate the time we spent together.

Similarly, during my time as a PhD student of the group, I spent time during my tea breaks to distract some master students from working on their thesis projects. I remember chatting with Martijn Dwars, Maarten Sijm, Taico Aerts, Chiel Bruin, Bram Crielaard, and more. We would talk about their work, sometimes mine, and plenty of the time about other things in life. I ended up playing an old video game with Maarten, Taico, and Chiel: Age of Empires II. We're still friends, we still play the game every other week.

I was also lucky to get to supervise two great master students for their thesis projects: Toine Hartman and Matthijs Bijman. Both Toine and Matthijs were wonderful, motivated students whom I enjoyed working with. I reference their theses in the conclusion of this dissertation.

Apart from a video game now and again, I have played many many board games over the last years. The main culprit who got me into board games is Christoph Lofi. I met this quirky professor of the Web Information Systems research group by virtue of working on the same floor, and we became friends over the many Wednesday night board game sessions at his house. We still play once or twice a month, still with a mixture of people that we rope in with promises of a shared meal and a fun evening.

Apart from playing board games at Christoph's, I've also done the same with friends from the PL group: Gabriël Konat, Daniël Pelsmaeker, and Jesper Cockx. We even managed to finish the entire campaign game of Gloomhaven including the Forgotten Circle which finished the story on the Gloom.

As if I didn't have enough of my evenings taken up by complex board games, Danny Groenewegen invited me into his old group of university friends to play Magic the Gathering: a trading card game that can be arbitrarily complicated as you build your own deck of card and each card can change the rules of the game. Gabriël and Daniël joined sometimes as well, but the new faces I got to know were those of Ruben Wieman, Boaz Pat-El, Joost Heijkoop, Michel Deconinck, and Daniël van Gelderen. I'm very grateful for Ruben, who helped me out during the first summer of the COVID-19 pandemic. I was still living in student housing, and Ruben offered me the option to work from the living room of his house, on the other side of the room from where he was working from home.

Over the years, before and after the pandemic, I managed to learn the card game we played on weekend days. I would borrow the decks of cards of the other players, and marvel at the artwork, the strange mechanics, and the clever combinations they came up with. I managed to avoid buying my own cards and designing my own decks for a long time. But at some point, after many years, I found myself trying my hand at a deck design. It did not take long for the other people in our group to collude and give me some cards for my birthday, to get me started with the hobby. It worked, I now have two decks of my own design and I am testing new designs sometimes.

Speaking of the time I spent in student housing: I was lucky to be able to even get some student housing in Delft at the time. Daco Harkes was a fellow PhD student in our research group at the time who lived in a 2 bedroom, shared kitchen and bathroom student housing situation when his housemate left. He asked me if I was interested the place, so I had a look and a chat with him and ended up taking the room. It is funny how Daco wasn't necessarily a person I would have normally befriended, but he was a very good housemate who could communicate clearly and was easy going. We ended up eating many an evening meal together, and had deep conversations about life, politics, religion, and more. I'm grateful for the opportunity Daco gave me to live in Delft, which allowed me to expand my social circles in all the ways I've already mentioned above and will expand on for one more page.

I joined local badminton club 'Swetiday' in Delft after playing there for a few Fridays with some of my colleagues from university. We originally picked the club because they offered the option to pay per evening and we didn't know how many times we might go and play. I ended up being the only one playing regularly, and so I joined the club as a member and got to know many new people there. Although none of us knew at first, through this club I met the father of one of the master students doing his thesis project with our PL group. Funny how those things go, it fits the argument that Delft feels like a small town.

During the pandemic, a group of friends that know each other through badminton started a 'bootcamp' group to keep exercising, outside, at an appropriate distance from one another. When I learned about this group I expressed my interest in joining and was invited. It turned out that during normal times, this group of friends would also go hiking or spend winter sports holidays together. I had accidentally joined more than an opportunity to do some sports, I joined a wonderful group of friends. By now I've been invited to people's homes, met their families, and been along for a long weekend of hiking in Limburg. I feel fortunate to have met more wonderful friends, who have welcomed me into their group.

Apart from all the new friends I made as I moved to Delft for my PhD, I still have some friends from before then too. As I mentioned, I studied in Delft for my bachelor's and master's too, but commuted from my parent's house at the time. During those years I met the jolly group of friends we now call Aajeto, after the time most of them lived together in Delft: Jelle Licht, Felix Akkermans, Maarten van Beek, and Thomas van Helden. I fondly recall the dinners we made and ate together, paranoid fiction films we watched, that night we stayed up late and tried to write mods for Call of Duty 1, and then played them, and other times where I slept on the couch because I'd stayed too long and the bus home had stopped for the night.

During my master's degree I went on an internship in the US, where I was welcomed by a fairly international team of people. These were all very wonderful colleagues who invited us interns to barbecues and went along to touristy things we would do as newly arrived interns. I met a particular good friend there in Alexander Weld, who I kept in touch with, and have continued to meet up with whenever we were relatively close to each other, for example due to my being at a conference in the US, or him (temporarily) working at an office in Europe. Alex got me addicted to another video game, called Duck Game, and we've had many laughs over the silly ways we messed up that in fast-paced game, and have had many (for me) late night conversations online where we'd planned to play that game but instead just mostly talked.

I could mention more friends I've spent time with during my PhD, like Vivian van der Werf (who I've known since secondary school!), or the student association I joined, but this preface is taking on ridiculous proportions already.

Have I mentioned how one my teachers in secondary school told me that one of my weaknesses is summarisation, keeping things short and to the point? Anyway, the connection between these friends, some of which I don't see regularly through university or sports, is mostly that I wish I could see them more often. Life can get arbitrarily busy, and I often wish there were more hours in a day. I guess I should have studied theoretical physics after all.

Jeff Smits
16th August 2023
Delft

English Summary

Computers execute software to do the tasks we expect from them. This software is written by human beings, we call this programming. The most common way to program is by writing text in a programming language. A programming language is very structured so we can be precise, but ultimately these languages are still for humans to read and write. In order to execute the written program, we need to translate it to a list of tiny instruction steps that the hardware of the computer can execute. This translation is also automated with software. The most common forms this software takes is (1) interpreters that execute a program live as they read it, or (2) compilers that translate the entire program for later execution.

Interpreters and compilers are tools of the domain of Programming Languages (PL). Apart from interpreters and compilers, there is more support software available around programming languages. This includes smart text editors, program analysis, running-program observers, etc. The requirements for PL tools are high: they should not get in the way when used to create software. In particular, they should support useful features, be fast enough in interaction, and not make mistakes.

Given these requirements, it is not a simple task to make PL tools. In an effort to make it easier to create PL tools, Language Workbenches (LWBs) were created: a suite of tools specifically for creating PL tools.

In this dissertation, you can find several improvements I made to a particular language workbench. I have—in multiple ways—sped up the *language development cycle* in this workbench: in terms of improved development, feedback, and execution speed.

Throughout my research, I have worked on and in the Spoofax language workbench, a research language workbench used for programming language research at TU Delft. Spoofax splits up the specification of programming languages into different domains, and captures each of those domains in a meta-language. For example, to describe the structure of the text of a programming language, Spoofax uses a formalism based on context-free grammars, extended with different useful features, which is called the Syntax Definition Formalism 3 or SDF3 for short. Similarly, there are meta-languages for the description of names, references and types; for what it means to execute a program; for defining assumptions and behaviour by example for testing purposes; and for transforming programs, which is a catch-all, but still a fairly high-level language. This language for transforming programs, called Stratego, is particularly relevant to this dissertation.

Contributions. Firstly, we introduce a new meta-language specialised in control- and data-flow analysis: FlowSpec. FlowSpec improves the development speed of programming languages in Spoofax, and the feedback in Spoofax and in the PL tools generated by Spoofax.

Secondly, we improve the compilation speed of Stratego on successive compilations with an incremental compiler. This compiler improves the speed at which you receive feedback inside Spoofax on changes to a Stratego program, and the speed at which you can see the results of tests and other short program executions after a change.

Thirdly, we add a gradual type system to Stratego to improve the feedback that can be given without executing Stratego programs. A gradual type system does not require a user of Stratego to add types to their program, but if they choose to, the gradual type system will be able to reason about the parts of the program that are typed, and give certain errors at compilation time instead of run time.

Finally, we develop a pattern matching optimisation that work for Stratego's pattern matching. This improves the execution speed of Stratego programs. Since all PL tools created in Spoofax include at least some of those Stratego programs, this also speeds up the execution of the Spoofax meta-languages themselves.

Nederlandse Samenvatting

Computers voeren software uit om de taken uit te voeren die we van hen verwachten. Deze software is geschreven door mensen, we noemen dit programmeren. De meest gebruikelijke manier om te programmeren is door tekst in een programmeertaal te schrijven. Een programmeertaal is erg gestructureerd, zodat we exact kunnen zijn, maar uiteindelijk zijn dit nog steeds talen voor mensen om te lezen en schrijven. Om het geschreven programma uit te voeren, moeten we het vertalen naar een lijst met kleine instructiestappen die de hardware van de computer kan uitvoeren. Ook deze vertaling is softwarematig geautomatiseerd. De meest voorkomende vormen van deze software zijn (1) interpreters die een programma uitvoeren terwijl ze het lezen, of (2) compilers die het hele programma vertalen voor uitvoering op een later moment.

Interpreters en compilers zijn gereedschap, tools, uit het domein van programmeertalen (PL). Naast interpreters en compilers is er meer ondersteunende software beschikbaar rond programmeertalen. Zo zijn er slimme tekstverwerkingsprogramma's, programma-analyse tools, waarnemers voor draaiende programma's, enz. De eisen aan PL-tools zijn hoog: ze mogen niet in de weg zitten bij het maken van software. Daarom moeten ze handige functies ondersteunen, snel genoeg zijn in interactie en geen fouten maken.

Gegeven deze vereisten is het geen eenvoudige taak om PL-tools te maken. In een poging om het maken van PL-tools gemakkelijker te maken, zijn taalontwikkelomgevingen gemaakt: een softwarepakket specifiek voor het maken van PL-tools.

In dit proefschrift vind je verschillende verbeteringen die ik heb ontwikkeld voor een specifieke taalontwikkelomgeving. Ik heb op meerdere manieren de *taalontwikkelingscyclus* in deze ontwikkelomgeving versneld: zowel ontwikkelings-, feedback- als uitvoeringssnelheid.

Tijdens mijn onderzoek heb ik gewerkt aan en in de taalontwikkelomgeving Spoofax, een onderzoeks-taalontwikkelomgeving die wordt gebruikt voor programmeertalenonderzoek aan de TU Delft. Spoofax splitst de specificatie van programmeertalen op in verschillende domeinen en vangt elk van die domeinen in een meta-taal. Om bijvoorbeeld de structuur van de tekst van een programmeertaal te beschrijven, gebruikt Spoofax een formalisme gebaseerd op contextvrije grammatica's, uitgebreid met verschillende handige functies, en dit wordt de Syntax Definition Formalism 3 of kortweg SDF3 genoemd. Zo zijn er meer meta-talen: voor de beschrijving van namen, referenties en typen; voor wat het betekent om een programma uit te voeren; voor het definiëren van aannames en gedrag via voorbeelden, voor test-doeleinden; en voor het transformeren van programma's, wat een grote laatste stap is, maar nog steeds in een taal van redelijk hoog niveau wordt gevangen. Met name deze taal voor het transformeren van programma's, genaamd Stratego, is relevant voor dit proefschrift.

Bijdragen. Ten eerste introduceren we een nieuwe meta-taal die gespecialiseerd is in besturings- en gegevensstroom analyse: FlowSpec. FlowSpec verbetert de ontwikkelingssnelheid van programmeertalen in Spoofax en de feedback in Spoofax en in de PL-tools die door Spoofax worden gegenereerd.

Ten tweede verbeteren we de compilatiesnelheid van Stratego voor regelmatig achtereenvolgende compilatie met een incrementele compiler. Deze compiler verbetert de snelheid waarmee je binnen Spoofax feedback krijgt over wijzigingen aan een Stratego-programma, en de snelheid waarmee je na een wijziging de resultaten van tests en andere korte programma's kunt zien.

Ten derde voegen we een geleidelijk typesysteem aan Stratego toe om de feedback te verbeteren die kan worden gegeven zonder Stratego-programma's uit te voeren. Een geleidelijk typesysteem vereist niet dat een gebruiker van Stratego types aan zijn hele programma toevoegt, maar als diegene ervoor kiest om enkele types toe te voegen, kan het geleidelijke typesysteem redeneren over de delen van het programma dat types heeft, en een bepaalde fouten detecteren tijdens het compileren in plaats van tijdens het uitvoeren.

Ten slotte ontwikkelen we een optimalisatie voor patroonvergelijking die compatibel is met Stratego's patroonvergelijking. Dit verbetert de uitvoeringssnelheid van Stratego-programma's. Gezien alle PL-tools die in Spoofax zijn gemaakt in ieder geval voor een deel bestaan uit Stratego-programma's, versnelt dit ook de uitvoering van de meta-talen van Spoofax zelf.

Table of Contents

Chapter 1

Introduction

Computers of all kinds execute software to function. These computers can be desktop and laptop computers, but also mobile phones (smart or not), graphing calculators, (smart) appliances, and so on. The hardware has general and specific (computation) abilities, and connects different parts, but it is the software that controls it all. That software is written by human beings, in an effort called programming.

Programming can be done in different ways, but one of the most prevalent ways is still to write text in a programming language (Bissyandé et al. 2013). Such programming languages are very structured but ultimately still languages for humans to read and write (Fowler 1999, p. 15). In order to execute the program written in a programming language we need a way to translate it to tiny instruction steps that the hardware is capable of executing. This translation step is something we automate with another program. Translation programs are typically either interpreters—these take in program text, translate as they read and immediately execute it—or compilers, which take in program text and translate it into an executable program which can be run later.

These interpreters and compilers are tools of the domain of Programming Languages (PL). Other such tools include smart program text editors ('editors'), program (text) analysis and manipulation tools ('refactoring tools'), live program observers ('debuggers'), and Integrated Development Environments (IDEs) that combine all of the above. Since PL tools are software itself, we can use them to develop even better PL tools. But the requirements for useful tooling are high: PL tools should not be in the way of making the actual software. They could be in the way if they do not support the right features, are too slow in interaction, or if they make mistakes (J. Nielsen 1993, ch. 5; Yang et al. 2011).

And so, making new PL tools is not a simple task. Therefore there have been efforts for a long time already to make creating PL tools easier (Schorre 1964). Language Workbenches (LWBs) are suites of tools specifically designed for making PL tools (Fowler 2005). *This dissertation is about the various improvements made to a particular language workbench, and how the ideas behind those improvements may be applied elsewhere as well.* Before discussing the contributions of this dissertation, we will first take a closer look at LWBs, and their limitations.

1.1 Language Workbenches

To understand language workbenches, we must first look at programming languages a little more closely. There are roughly two important groups of programming languages: General-Purpose Languages (GPLs) and Domain-Specific Languages (DSLs).

General Purpose Languages. As the name suggests GPLs are for a wide public of programmers to use for any purpose. Learning a GPL pays off because it is widely usable. These languages are generally popular, and the popular ones have a network effect. Because many people know a particular GPL, companies can more easily adopt it, as there are enough people to hire to work on software written in the language. And the developers of a popular GPL are also likely to be funded to further develop the language, ecosystem of tools, and reusable code collections (libraries). The large group of users are likely to contribute (to) libraries, some of the tooling, and at least discussions if not implementations of new language features (Kogut and Metiu 2001).

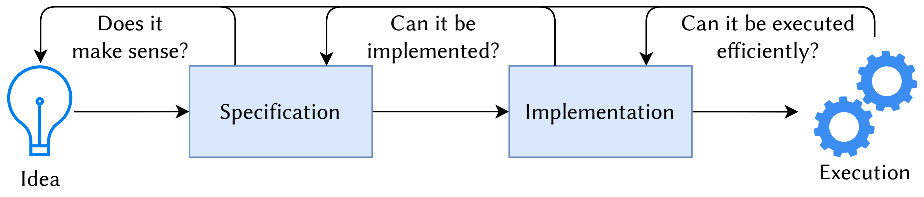
A number of popular GPLs typically dominate particular markets (Bissyandé et al. 2013; EDC 2022). The network effect that makes them useful, also makes them hard to supplant. Even if the GPL itself has some poorly designed features, popularising an improved GPL can be difficult due to the lack of tools, libraries and network of users. There is also a trade-off between improving on the old ideas of current GPLs and the cost that brings to learn a new language. If a new GPL is too different from current ideas, that makes it harder to attract users to get the network effect, which is necessary to make the language truly successful. If a new GPL is too familiar, it might not be enough of an improvement for people to make the switch from an existing popular GPL.

Domain Specific Languages. The DSL is a different type of programming language. It is designed specifically for software within a certain domain. That domain's terminology and knowledge is (at least partially) integrated into the language (van Deursen, Klint and J. Visser 2000; Fowler 2010). This makes each DSL for a different domain very different and perhaps harder to learn for anyone unfamiliar with the domain in question. Because a DSL typically serves a smaller market than GPLs do, there is a smaller group of programmers that use the language, which lessens the network effect.

Thankfully, there is less of a dependence on reusable code libraries in DSLs. This is because the integration of domain terms and knowledge into the language is doing much of the heavy lifting. Viewed from a different perspective though, that means that the language designers and implementors need to actually provide that powerful functionality. Next to that extra work, programmers are spoiled by the mature tooling of GPLs and want this for a DSL too.

Language Workbenches. And so we come upon LWBs again. We have now seen that to create a new GPL that can compete with existing ones, we need to provide the right mix of old and new features, which may require some experimentation. Making that experimentation cheap is important to explore a feature space. Similarly, a DSL designer will need to experiment with how to integrate the domain knowledge and concepts into the language. We have also seen that for both GPLs and DSLs it is important to provide the tooling that programmers have gotten used from existing languages.

As it turns out, the implementation of programming languages and their tools for both GPLs and DSLs are based on common techniques. These are



☙ **Figure 1.1** The Language Development Cycle(s).

captured in LWBs to make the creation and exploration of programming languages and their tools easier. This makes it possible to explore different ideas and their interaction within a programming language.

There are three common approaches to LWBs. The first is to use feature modelling from the object-oriented programming world to create a rich semantic model of the programming language. This is approach is used by Jetbrains MPS (Pech 2021), MontiCore (Grönniger et al. 2008), and Gemoc (Combemale, Barais and Wortmann 2017). The second is a more operational approach, where a LWB uses a single powerful meta-language to express the programming language implementation. This is used by the Rascal Meta-Programming Language (Klint, van der Storm and Vinju 2009), and by reference attribute grammar systems like JastAdd (Ekman and Görel Hedin 2007a) and Silver (Van Wyk et al. 2010). Finally, the third approach is to split up a programming language specification into different domains and capture each with a meta-DSL. This is the approach of the ASF+SDF Meta-Environment (van den Brand, van Deursen et al. 2001) and the Spoofax Language Workbench (Kats and E. Visser 2010).

To summarise what LWBs do, Figure 1.1 depicts the Language Development Cycle(s). These are the feedback cycles during language development. When we move from a language development idea to a specification of the language, we write a human-readable definition of the language at a high level of abstraction. This allows us to reflect on the idea to see if it fits into the overall design. The next step is to implement this specification and see if the operational details are achievable. Finally once this implementation exists and can be executed, we can see if the execution is efficient enough. In each step, the feedback cycle informs either the previous step, or even further back, depending on how fundamental the problem is. LWBs provide facilities to create the implementation, or even a combination of specification and implementation. These facilities are there to make (specification and) implementation easier to express and provide feedback during the development.

1.2 Limitations of Language Workbenches

While LWBs are great in theory for the exploration of programming language designs and the creation of PL tools, in practice there are some problems. It may still be very hard to implement particular design ideas, and the resulting PL tools may be too slow to be useful.

Therefore, my main research question is:

How can the Language Development Cycle be sped up?

There are three aspects to this 'speed' that we will consider in this dissertation:

1. The *development* speed with which we can go from a PL idea to an executable specification of that idea. This is influenced by the expressiveness and level of abstraction that the LWB provides.
2. The *feedback* we receive from the language workbench on our specifications. Fast and good quality feedback speeds up the process of specifying and implementing a PL idea.
3. The *execution* speed that our resulting PL tools run at. In the end, for a derived PL tool to be useful, it must be sufficiently responsive to user interaction.

1.3 Research Methods

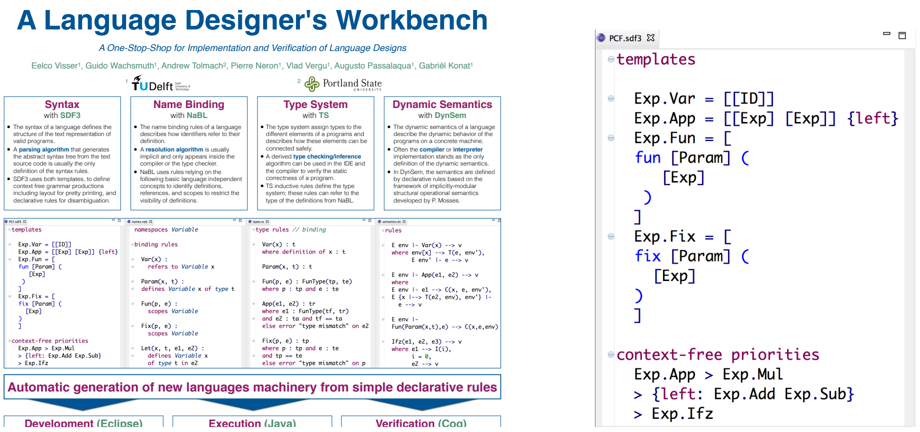
To answer the research question, taking all three aspects of speed into account, we take an approach sometimes called programming systems research (E. Visser 2021). This form of research emphasises the development of research software to the point of practically usable tools, because that is the true test for the underlying ideas. To put it more concretely: developing research software allows us to test if our methods indeed speed up the language development cycle.

One of the ways we test our methods is by applying them to case studies (Runeson and Höst 2009; Flyvbjerg 2006). For example, in Chapter 2 we evaluate our new specification language through case studies on two real programming languages, an academic one we use ourselves and an industrial DSL. Another form of evaluation that we do is to benchmark our prototypes. For example, the controlled experiments that benchmark our prototype in Chapter 2 are described in Sections 2.6.11 and 2.7.4.

In fact, each of the chapters has a prototype implementation that we use for evaluation. To facilitate replication studies of all our experimental results during the PhD we have published artefacts with *every* publication that had an evaluation based on experiments. When artefact evaluation was available we submitted these artefacts for review. All reviewed artefacts were deemed functional according to the standards (ACM 2020) set out by the Association for Computing Machinery (ACM).

1.4 Research Context: Spoofax and Stratego

The programming systems research that we present in this dissertation is done within the context of the Spoofax Language Workbench. We contribute ideas for language design and implementation, instigated by, motivated by, and tested in Spoofax. Spoofax is one of the LWBs we mentioned earlier, which



A Language Designer's Workbench

A One-Stop-Shop for Implementation and Verification of Language Designs

Eelco Visser[1], Guido Wachsmuth[1], Andrew Tolmach[2], Pierre Neron[1], Vlad Vergu[1], Augusto Passalaqua[1], Gabriël Konat[1]

[1] TUDelft Delft University of Technology [2] Portland State University

Syntax with **SDF3**

- The syntax of a language defines the structure of the text representation of valid programs.
- A **parsing algorithm** that generates the abstract syntax tree from the text source code is usually the only definition of the syntax rules.
- SDF3 uses both templates, to define context free grammar productions including layout for pretty printing, and declarative rules for disambiguation.

Name Binding with **NaBL**

- The name binding rules of a language describes how identifiers refer to their definition.
- A **resolution algorithm** is usually implicit and only appears inside the compiler or the type checker.
- NaBL uses rules relying on the following basic language independent concepts to identify definitions, references, and scopes to restrict the visibility of definitions.

Type System with **TS**

- The type system assign types to the different elements of a programs and describes how these elements can be connected safely.
- A derived **type checking/inference** algorithm can be used in the IDE and the compiler to verify the static correctness of a program.
- TS inductive rules define the type system; these rules can refer to the type of the definitions from NaBL.

Dynamic Semantics with **DynSem**

- The dynamic semantics of a language describe the dynamic behavior of the programs on a concrete machine.
- Often the **compiler** or **interpreter** implementation stands as the only definition of the dynamic semantics.
- In DynSem, the semantics are defined by declarative rules based on the framework of implicitly-modular structural operational semantics developed by P. Mosses.

PCF.sdf3

```
templates

  Exp.Var = [[ID]]
  Exp.App = [[Exp] [Exp]] {left}
  Exp.Fun = [
  fun [Param] (
     [Exp]
   )
  ]
  Exp.Fix = [
  fix [Param] (
    [Exp]
  )
  ]

context-free priorities
  Exp.App > Exp.Mul
  > {left: Exp.Add Exp.Sub}
  > Exp.Ifz
```

names.nab

```
namespaces Variable

binding rules

  Var(x) :
    refers to Variable x

  Param(x, t) :
  defines Variable x of type t

  Fun(p, e) :
    scopes Variable

  Fix(p, e) :
    scopes Variable

  Let(x, t, e1, e2) :
    defines Variable x
    of type t in e2
```

types.ts

```
type rules // binding

  Var(x) : t
  where definition of x : t

  Param(x, t) : t

  Fun(p, e) : FunType(tp, te)
  where p : tp and e : te

  App(e1, e2) : tr
  where e1 : FunType(tf, tr)
  and e2 : ta and tf == ta
  else error "type mismatch" on e2

  Fix(p, e) : tp
  where p : tp and e : te
  and tp == te
  else error "type mismatch" on p
```

semantics.ds

```
rules

  E env |- Var(x) --> v
  where env[x] --> T(e, env'),
        E env' |- e --> v

  E env |- App(e1, e2) --> v
  where
  E env |- e1 --> C(x, e, env'),
  E {x |--> T(e2, env), env'} |-
    e --> v

  E env |-
  Fun(Param(x,t),e) --> C(x,e,env)

  Ifz(e1, e2, e3) --> v
  where e1 --> I(i),
        i = 0,
        e2 --> v
```

Automatic generation of new languages machinery from simple declarative rules

Development (Eclipse) | Execution (Java) | Verification (Coq)

PCF.sdf3

```
templates

  Exp.Var = [[ID]]
  Exp.App = [[Exp] [Exp]] {left}
  Exp.Fun = [
  fun [Param] (
     [Exp]
   )
  ]
  Exp.Fix = [
  fix [Param] (
    [Exp]
  )
  ]

context-free priorities
  Exp.App > Exp.Mul
  > {left: Exp.Add Exp.Sub}
  > Exp.Ifz
```

Figure 1.2 Part of a Spoofax poster accompanying a vision paper (E. Visser, Wachsmuth et al. 2014), and the SDF example zoomed-in.

takes the approach to split up a programming language specification into different domains, and capture each with a meta-DSL. Our research software includes a new meta-DSL called FlowSpec, and improvements to an existing meta-DSL called Stratego.

1.4.1 *The Spoofax Language Workbench*

Spoofax is a language workbench that takes the approach of splitting up a programming language specification into different domains, and capture each of those domains with a meta-DSL. The Spoofax project is an active research project, that is steadily improving over the years. We'll have a look at Spoofax as I got to know it at the time, illustrated by Figure 1.2. When I started on my PhD programme in 2016, the Spoofax language workbench had components in the following five domains.

Grammars describe the textual representation of a programming language. The Syntax Definition Formalism (SDF) captures the domain of grammars. The parser generated from an SDF specification can turn text that matches the grammar into an Abstract Syntax Tree (AST). The labels on the grammar rules form the labels of the tree, and each node in the tree has the corresponding grammar sort as its type. These ASTs are the program representation for other components in the workbench.

SDF3 was the version under active development. For example, de Souza Amorim, Erdweg et al. (2016) added (syntactic) code completion, derived from the specification by writing the grammar as a template. These templates can be seen on the right in Figure 1.2, where an expression grammar is defined. Each grammar rule for expression `Exp` is followed by a dot and a label. On the right-hand side of the equals sign is the template with program text, and bracketed sorts interspersed, which together describes the tree shape of the program text.

Static Semantics is the meaning of a program without executing it. The Name Binding Language (NaBL) captures name rules (definition and resolution of names), while the Type System Language (TS) captures simple type systems. These languages refer to each other's concepts and were a pair. Together, they capture part of the domain of static semantics.

This is illustrated by the poster in Figure 1.3 in the left two examples. The left of the two examples describes where variables are defined, where they are referenced, and what their lexical scope is. It also speak of types where they are present in the program. Each of these rules matches on an AST node with a label from the grammar. The type rules of TS on the right also match on AST nodes, compute types for those nodes, and refer to types of the definition of variables.

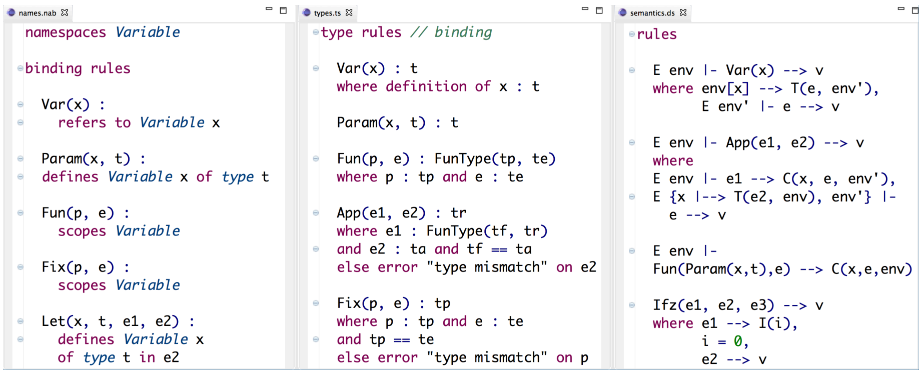
At the time, NaBL2 was under active development, based on the—at the time—new idea of scope graphs (Néron et al. 2015). Scope graphs can express more complex name and type patterns that were not expressible in NaBL and TS.

Notably, while Spoofax has captured name and type analysis in meta-DSLs, control- and data-flow analysis is not part of that. Control flow relates to the order in which parts of a program are executed. Programs test data according to certain conditions to decide what to do next. A common form of this is the `if-then-else`, which checks *if* a certain condition holds and in that case executes the *then* code, while instead executing the *else* code if the condition does not.

Data flow analysis builds on top of control flow analysis to predict the approximate values the program will compute, or the approximate behaviours of the program, when it is executed. This is used to provide static guarantees, such as identifying and disallowing code that will never be executed; and to provide insights that can be used to optimise a program, such as moving some code to a location where it is executed less if the result is predicted to commonly give the same result.

Dynamic Semantics is the meaning of a program when executed. The DynSem language captures part of the dynamic semantics domain. DynSem specifications use rules to do computation steps on programs, partially inspired by Modular Structural Operational Semantics (Mosses 2004). From these DynSem specifications interpreters can be generated (Vergu, Néron and E. Visser 2015). With dynamic semantics specified, we can also compare that it agrees with the static semantics of the same programming language. This is typically done by writing a mathematical proof that relates the two. In mathematical proofs, details matter, and in some cases we may want a small-step semantics, which refines programs in small steps towards values. While this is possible in DynSem, there is a preference for big-step semantics which computes values directly from programs (Vergu, Néron and E. Visser 2015, p. 366). This makes it easier to generate interpreters with reasonable performance, as opposed to small-step semantics, which needs optimisations (such as refocussing (Danvy and L. R. Nielsen 2004)).

The right-most example of Figure 1.3 shows some DynSem rules. The `E env`


names.nab

```
namespaces Variable

binding rules

  Var(x) :
    refers to Variable x

  Param(x, t) :
  defines Variable x of type t

  Fun(p, e) :
    scopes Variable

  Fix(p, e) :
    scopes Variable

  Let(x, t, e1, e2) :
    defines Variable x
    of type t in e2
```

types.ts

```
type rules // binding

  Var(x) : t
  where definition of x : t

  Param(x, t) : t

  Fun(p, e) : FunType(tp, te)
  where p : tp and e : te

  App(e1, e2) : tr
  where e1 : FunType(tf, tr)
  and e2 : ta and tf == ta
  else error "type mismatch" on e2

  Fix(p, e) : tp
  where p : tp and e : te
  and tp == te
  else error "type mismatch" on p
```

semantics.ds

```
rules

  E env |- Var(x) --> v
  where env[x] --> T(e, env'),
        E env' |- e --> v

  E env |- App(e1, e2) --> v
  where
  E env |- e1 --> C(x, e, env'),
  E {x |--> T(e2, env), env'} |-
    e --> v

  E env |-
  Fun(Param(x,t),e) --> C(x,e,env)

  Ifz(e1, e2, e3) --> v
  where e1 --> I(i),
        i = 0,
        e2 --> v
```

Figure 1.3 The NaBL, TS and Dynsem examples from the poster.

environment maps names to values, much like how in static semantics a name is mapped to a type. Pieces of a program are matched as AST nodes again, with long arrows that show the step to values.

Rewriting is a general technique used in PL tools. The Stratego term rewriting language (Hemel, Kats et al. 2010; Olmos and E. Visser 2005), is not specific to the definition of static or dynamic semantics rules. The rewrite rules of Stratego still match on AST nodes, and turn them into other AST nodes. This can be used for many things, like desugaring (simplifying the AST to make it easier to work with), name resolution (static semantics), code generation (an indirect form of dynamic semantics), including optimisation, and more. A special feature of Stratego called dynamic rewrite rules (Bravenboer, van Dam et al. 2006) was developed that could do a form of control- and data-flow analysis, although it is not used very often for this purpose. At the time, Stratego was not under active development; it was the mature glue language of Spoofax as well as the escape hatch for things that a meta-DSL could not express.

Testing checks assumptions, finds mistakes, and documents behaviour. Testing—of parsing, static semantics such as name resolution and types, and execution of Stratego rewrite rules—was done in the Spoofax Testing Language (SPT) (Kats, Vermaas and E. Visser 2011). Any serious (software) project, including PL tools defined in Spoofax, benefit from tests. SPT provides some convenient ways to test PL specific tasks that are extracted from a specification in Spoofax. For example, we can write a small program in the language we defined, marking a name at definition and reference site, and tell SPT to test that the name resolves from the reference to the definition.

During my PhD, NaBL version 2 was developed further, and then replaced by Statix. DynSem's successor—Dynamix—was also proposed during this time, and is still under development.

1.4.2 *The Stratego Term Rewriting Language*

Stratego is a term rewriting language that supports program transformation with programmable rewriting strategies (E. Visser, Benaissa and Tolmach 1998). The shapes of terms are described by algebraic signatures, which are similar to algebraic data types in Functional Programming (FP), or regular tree grammars. In fact, most of the signatures we use in Stratego are generated from an SDF specification. Stratego, within Spoofax, mostly rewrites terms that represent the AST defined in SDF.

Rules. Rewriting is done through rewrite rules, which match an AST pattern and produce an AST pattern. This is illustrated in Figure 1.4, with `cnf-rule`. In Stratego, we do not have to explicitly call rewrite rules on the children of the AST node we matched on though, like we might in functional programming. In term rewriting, we instead use a rewrite strategy to apply the rewrite rules everywhere in the tree. For example, we can apply rules top-down, bottom-up, repeat them until there is nothing left to rewrite, etc. In the example, `innermost` applies the rewrite rules bottom-up, repeating them exhaustively.

Strategies. Because Stratego has programmable strategies, we can choose when, where and what strategy to use. We can also write our own strategies, either by referring to other strategies (e.g. `innermost`), or by using primitives for *generic traversal* (e.g. in `bottomup`, where `all` is a primitive). These are named traversals because we specify how to traverse the tree, and generic because we do not necessarily care about the labels on the tree nodes (that is what the rewrite rules are for). The downside of these powerful generic traversal primitives is that they cannot easily be checked by a static type system. The initial solution for Stratego was to use a minimal dynamic type system, without checks at run-time.

Modules. Rewrite rules with the same name can be written across different files free-form, however the user wants, and are merged together by the compiler automatically. In order to provide such a flexible module system, Stratego's has a so-called whole-program compiler. The compiler resolves all imports to other files itself, finds the entire program, and compiles it all at once. In contrast, some compilers will compile each file separately, perhaps needing some limited information from other files, or postponing cross-file checking to a separate tool. While the whole-program compiler is convenient in use, the cost is that as Stratego programs grow larger, the compiler takes longer.

1.5 Contributions

Within the context of Spoofax in general, and Stratego in particular, we present our contributions.

1.5.1 A Meta-DSL for Control- and Data-flow Analysis

▷ To improve the *development speed* in Spoofax, and the *feedback* in Spoofax and PL tools developed in Spoofax, we introduce a new meta-DSL, which is specialised towards control- and data-flow analysis: FlowSpec. This DSL fills a gap in Spoofax, as this kind of static analysis that was not commonly

```
rules
  cnf-rule: Or(And(a1, a2), o) -> And(Or(a1, o), Or(a2, o))
  cnf-rule: Or(o, And(a1, a2)) -> And(Or(o, a1), Or(o, a2))
strategies
  cnf = innermost(cnf-rule)
  innermost(s) = bottomup(try(s; innermost(s)))
  bottomup(s) = all(bottomup(s)); s
```

Figure 1.4 Conjunctive normal form through rewriting in Stratego.

implemented. ◁

FlowSpec captures the domain of control- and data-flow analysis through explicit use of domain terms from control-flow graphs and data-flow equations. The control-flow graph rules are syntax directed and composed by referring to the surrounding control-flow graph. The data-flow equations match on an edge of the control-flow graph and propagate data-flow information from one node to the other. The final piece of the puzzle is that a data-flow analysis is not only typed, but has an associated lattice instance, which provides the operation to use where control-flow merges, and provides a termination guarantee for the analysis.

FlowSpec has been evaluated with case studies of common analyses written for Stratego, and for GreenMarl, an industrial DSL for graph analytics (Hong, Chafi et al. 2012). Each of these case studies includes a performance measurement with benchmarks.

In summary, FlowSpec fills the gap of control- and data-flow analysis in Spoofax, by providing a high-level, declarative DSL for specification of those analyses.

1.5.2 An Incremental Compiler Through an Internal Build System

▷ To improve the *feedback* in Spoofax, we improve the compilation speed of Stratego with an incremental compiler. ◁

New meta-DSLs in Spoofax usually (partially) compile to Stratego. This results in a larger amount of Stratego code to compile, even as less Stratego code is actually written. Since Stratego has a relatively slow whole-program compiler, Spoofax language projects are getting slower to build as a consequence of the nice abstractions that the new meta-DSLs introduce.

If only Stratego could be incrementally compiled. Or even just separately compiled, so an old-fashioned build system could cache that for incrementality. But Stratego has language features that make it impossible to compile separately while staying compatible with the existing runtime.

Therefore our solution is to turn things around. We cannot put a separate compiler in an incremental build system. Therefore, we put the incremental build system inside the compiler. For the build system we use Pipelines for Interactive Environments (PIE), an incremental build system with support for dynamic dependencies (Konat, Erdweg and E. Visser 2018). We can find and track the complex dependencies between different pieces of Stratego code in PIE by pulling the Stratego compiler apart into smaller components, identifying pertinent information, and passing this to PIE. This results in an incremental compiler for Stratego that is now available in Spoofax. It is backward compatible with the existing runtime system and with code compiled with the whole-program compiler.

We evaluate our incremental compiler by replaying the recorded version control history of a large Stratego codebase: WebDSL (Groenewegen, Hemel et al. 2008). This shows that the majority of the last 200 commits take under 10 seconds to compile incrementally, whereas the entire project takes over 90 seconds to compile with the original whole-program compiler.

In summary, we present an incremental compiler for Stratego that remains backward compatible with the existing runtime system and other compiled Stratego code, by putting an incremental build system *inside* the compiler.

1.5.3 A Gradual Type System for an Existing Language With Generic Traversals

▷ To improve the *feedback* on Stratego programs in Spoofax, we add a gradual type system to Stratego. ◁

Stratego has always featured *signatures* that describe the shape of the trees it works on. These certainly look like types, but Stratego actually did very little enforcement of these types in Stratego code. This lack of enforcement could be leveraged to work with ill-formed trees sometimes, without having to specify their type. But quite commonly, the ill-formed trees are an accident, a type bug in the Stratego program.

Adding static types to Stratego would solve this problem, but there are two major roadblocks. The first is that Stratego is built on generic traversal primitives that elude static typing in their general form. The second is that there is a large codebase of useful Stratego code that makes excellent use of all the dynamically typed freedom that Stratego offers. We have yet to find a static type system that would support generic traversals and provide a cheap enough migration path for existing code.

But we do not have to find such a static type system. Gradual type systems allow the mixing of dynamically typed code and statically typed code (Siek and Taha 2006). A gradual type system gives us the migration path we need for existing code, and a way out of defining types for all of Stratego. And so, we present a gradual type system for Stratego. In our design, we take the new incremental compiler for Stratego into account, and minimise the extra information that it has to track. We also find that we can incorporate existing research on typed generic traversals in our type system. With a gradual type system, we can already express queries and collecting traversals (*type-unifying*). By adding a special *type-preserving* type, we can also write generic traversals with the same input and output type, that can use an overloaded rewrite rule.

To evaluate the gradual type system for the purpose of migrating existing code, we tested adding types to an existing Stratego codebase of 36 files containing 3235 lines of code. During this evaluation we added 74 type signatures to standard library strategies that are used in the code and 117 type annotations to the code of the project. The majority (85 out of 117) type annotations are a fully static type. Numerous type-related bugs were found, which we roughly classify and present examples of. This small study shows that the gradual type system is useful to guide the transition of code to a better type discipline.

In summary, we present a gradual type system to add statically checkable type discipline to Stratego. The type system can give types to many common patterns in Stratego code, and supports dynamically typed code, to keep existing code working.

1.5.4 Mapping Pattern Match Optimisations to a Core Language for Rewriting

▷ To improve the *execution speed* of PL tools produced by Spoofax, we add an optimisation for pattern matching in Stratego. Because Spoofax is itself defined in Spoofax, this also increases the speed at which Spoofax runs, and thus improves *feedback*. ◁

With the addition of an incremental compiler and gradual type system, Stratego is now a much nicer language to use. However, these are only improvements to the *development cycle* in Stratego. The execution of compiled Stratego code can still be improved.

In some ways, term rewriting resembles FP, so we were inspired to bring an optimisation from FP into the Stratego compiler: pattern match optimisation. There was some suspicion already that this can improve execution speed, and there was an old, broken attempt at pattern match optimisation in the Stratego compiler already. But the main problem with copying this idea from FP is that Stratego has a very different core calculus.

Pattern matching in Stratego is compiled down to so-called first-class pattern matching primitives. These primitives are even used directly in Stratego to great effect in certain code patterns. In the end though, these primitives still express a pattern matching that seems similar to pattern matching in FP. So we look for, and find a way to extract FP-like pattern matching from Stratego programs.

We make sure that the proposed optimisation is actually effective by testing its implementation on a corpus of Stratego programs. We test Stratego programs written in multiple styles to make these measurements representative, and these show that the optimisation does indeed improve the execution speed of Stratego programs, in some cases dramatically so.

In summary, by finding a way to translate Stratego's first-class pattern matching to an FP-like pattern matching, we were able to apply well-known pattern matching optimisation techniques from FP to Stratego.

1.6 Structure

The main chapters of this dissertation are based on four peer-reviewed publications. I am the first author and main contributor of these publications. In the publication for Chapter 5, the status of first author and main contributor is shared with Toine Hartman, who executed most of the scientific investigation as part of his master's thesis project (Hartman 2022). I was the direct supervisor of this master's thesis project and was involved in the scoping of the project, advising the methodology used in the project, brainstorming with

Toine about details of the work, reviewing the code for the prototype, and advising on the written thesis. The publication based on the thesis project was written primarily by me, and I extended the evaluation.

Since each main chapter is based on a stand-alone publication with distinct contributions, there is some redundancy, especially in the introduction sections. This was left in so each chapter can be read independently. The main chapters and their corresponding publication are as follows:

- Chapter 2 is an updated version of the COMLAN 2020 paper *FlowSpec: A Declarative Specification Language for Intra-Procedural Flow-Sensitive Data-Flow Analysis* (Smits, Wachsmuth and E. Visser 2020).
- Chapter 3 is an updated version of the Programming 2020 paper *Constructing Hybrid Incremental Compilers for Cross-Module Extensibility with an Internal Build System* (Smits, Konat and E. Visser 2020).
- Chapter 4 is an updated version of the SLE 2020 paper *Gradually typing strategies* (Smits and E. Visser 2020).
- Chapter 5 is an updated version of the SLE 2022 paper *Optimising First-Class Pattern Matching* (Smits, Hartman and Cockx 2022).

Finally, we end with a conclusion in Chapter 6 where we summarise our work, discuss the work in relation to the main research question, and discuss future work.

Appendix A is a supplement to Chapter 2, and Appendix B is an unpublished supplement to Chapter 4. Both are referenced in their respective chapters where relevant.

A note on citations. As you may have already noticed in the above citations, sometimes the citation includes an initial of an author. This is intentional, these are inserted automatically in cases where multiple authors with the same last name are in the bibliography.

Chapter 2

Language Parametric Control- and Data-Flow Analysis

Abstract. Data-flow analysis is the static analysis of programs to estimate their approximate run-time behaviour or approximate intermediate run-time values. It is an integral part of modern language specifications and compilers. In the specification of static semantics of programming languages, the concept of data-flow allows the description of well-formedness such as definite assignment of a local variable before its first use. In the implementation of compiler back-ends, data-flow analyses inform optimisations.

Data-flow analysis has an established theoretical foundation. What lags behind is implementations of data-flow analysis in compilers, which are usually ad-hoc. This makes such implementations difficult to extend and maintain. In previous work researchers have proposed higher-level formalisms suitable for whole-program analysis in a separate tool, incremental analysis within editors, or bound to a specific intermediate representation.

In this chapter, we present FlowSpec, an executable formalism for specification of data-flow analysis. FlowSpec is a domain-specific language that enables direct and concise specification of data-flow analysis for programming languages, designed to express flow-sensitive, intra-procedural analyses. We define the formal semantics of FlowSpec in terms of monotone frameworks. We describe the design of FlowSpec using examples of standard analyses. We also include a description of our implementation of FlowSpec.

In a case study we evaluate FlowSpec with the static analyses for GreenMarl, a domain-specific programming language for graph analytics.

Based on: Jeff Smits, Guido Wachsmuth and Eelco Visser (2020). 'FlowSpec: A Declarative Specification Language for Intra-Procedural Flow-Sensitive Data-Flow Analysis'. In: *Journal of Computer Languages* 57, 100924, page 39. DOI: 10.1016/j.cola.2019.100924

2.1 Introduction

F. Nielson, H. R. Nielson and Hankin define program analysis as follows: Program analysis offers static compile-time techniques for predicting safe and computable approximations to the set of values or behaviours arising dynamically at run-time when executing a program on a computer (F. Nielson, H. R. Nielson and Hankin 2005, p. 1). Data-flow analysis can answer questions such as if and when data in a variable is accessed, or if certain invariants hold on the data. Data-flow analyses are used to provide static guarantees in the form of compiler warnings and errors, to inform optimisations, to identify security problems, or problematic code style.

2.1.1 Uses Of Data-flow Analysis

Data-flow analyses may be part of the static semantics of a language. For example, in Java a final field in a class must be initialised for an object of that class by the end of its construction (Gosling et al. 2005, ch. 16). Since constructor code can have conditional control-flow, a data-flow analysis is necessary to check that all possible execution paths through constructors actually assign a value to the final field. For another example, the compiler for Rust gives warnings on code paths that are unreachable (Rust Project Developers 2018).

Data-flow analyses are commonly used to inform optimisations in compilers. Live variables analysis provides information on which variables will be used with their current value, which can be used by a form of dead code elimination called *dead store elimination* (Auslander and Hopkins 1982, p. 24). This optimisation removes assignments to variables which are not observed. Available expressions analysis identifies expressions that have already been computed, which can be used for *common subexpression elimination*.

In some compilers, and in separate tools, data-flow is used to identify security problems. A common approach is taint analysis, which can analyse where data from relevant sources flow. For example, a source of data could be untrusted data from user input. User input should not be used directly in the text of an SQL query, as this opens the possibility of SQL injections.

Data-flow analysis is also applied in code style tools that check for code patterns which are hazards to maintenance or likely to be a logic error. Examples are analyses such as a switch case in Java which has some code, but then falls through to the next case. Although a case that directly falls through is likely intentional, one that has some code may be missing a break statement. Another style lint, as these analyses are often called, is the definition of a non-final variable that is only assigned once. Both of these lints are part of the CheckStyle (Checkstyle team 2018) tool for Java.

2.1.2 Implementation of Data-flow Analysis

Data-flow analyses are important for the specification and implementation of programming languages and domain-specific languages (DSLs). However, they are expensive to implement, especially in a general purpose programming language. The compiler for GreenMarl (Hong, Sevenich and Lugt 2014), a graph analytics DSL from industry, requires more than 2000 lines of C++ code for a data-dependence analysis that takes the domain concepts of the language into account. Since DSLs typically have a relatively small audience, this is reflected in their development team size. The implementation cost of even the most common data-flow analyses can become prohibitive in such a situation.

Language workbenches aim to facilitate high-level language definition and generation of implementations, thereby improving the situation for DSL development. For example, the Spoofax language workbench (Kats and E. Visser 2010) provides declarative meta-languages for the concise specification of a programming language. An SDF3 (Vollebregt, Kats and E. Visser 2012) specification is used by Spoofax to generate a parser. An NaBL2 (van Antwerpen et al. 2016) specification of the static semantics of the language is used to generate a type checker.

The goal of this work is to provide the same benefits of concise, executable specification for data-flow analysis. In this chapter, we present FlowSpec, a specification language for intra-procedural, flow-sensitive data-flow analysis. FlowSpec is integrated in the Spoofax language workbench and makes use of the provided ecosystem. The analysis that is generated from FlowSpec consumes analysed abstract syntax trees, and turns these trees into a control-flow graph using the control-flow rules. The control-flow graph is then used as input for data-flow analysis, based on the data-flow rules in a FlowSpec specification. When the data-flow analysis requires names or types, these can be referenced directly in the specification.

We evaluate FlowSpec with specifications of analyses, and we present case studies in static analysis definitions for GreenMarl, an industrial DSL for high performance, concurrent graph analytics, and Stratego, a term transformation language.

In summary, the contributions of this chapter are:

- The language design of FlowSpec, a language parametric, domain-specific language for the definition of intra-procedural, flow-sensitive data-flow analysis.
- The formal semantics of FlowSpec in terms of Monotone Frameworks, a solid mathematical foundation that has been used for decades for sound approximation of data-flow information beyond sets.
- The implementation of FlowSpec, including the integration into the Spoofax language workbench, a fixed-point solving algorithm, and an adapted Strongly Connected Component (SCC) algorithm with extra ordering guarantees within the SCCs. The use of SCCs and their ordering is not novel, but we are not aware of a published algorithm that gives this directly.
- The evaluation of FlowSpec on the GreenMarl graph analytics DSL, which shows that the language can concisely and cleanly express analyses separately from the definition of transformations.
- The evaluation of FlowSpec on the Stratego term transformation language, which shows that the language can express interesting non-standard analyses

on more languages than a typical imperative language.

- The performance evaluation of FlowSpec on different size Stratego strategies, which show that the speed of FlowSpec is reasonable for use within an optimising compiler.

The paper this chapter is based on extends the initial SLE 2017 paper on FlowSpec (Smits and E. Visser 2017). We describe the FlowSpec design and implementation in more depth and provide evidence of its expressiveness by means of a significantly extended set of examples. We give a more complete definition of the syntax and semantics of the FlowSpec core language including its connection to name analysis using NaBL2. We describe the implementation of FlowSpec, including an adapted SCC algorithm and worklist algorithm, and discuss its integration in the Spoofax language workbench. We extend the case study of the application of FlowSpec to the specification of data-flow analyses for the Green-Marl data analytics DSL with more and more complete specifications of analyses, demonstrating that FlowSpec can be used to concisely define data-flow analyses that can be used to replace ad hoc implementations of these analyses in the Green-Marl compiler. We present a new case study, applying FlowSpec to the specification of reaching definitions analysis for the Stratego rewriting language. We evaluate the performance of FlowSpec analyses for GreenMarl and Stratego.

Outline. In the next section we discuss background on data-flow analyses and monotone frameworks. In Section 2.3 we introduce FlowSpec by example. We present the semantics of FlowSpec in Section 2.4. In Section 2.5 we describe the implementation of FlowSpec, both its integration into Spoofax and the independent solver algorithm. In Section 2.6 we present the first part of our evaluation of FlowSpec through data-flow analyses specified for the GreenMarl programming language. We present some benchmarks that show that these analyses are practically usable. In Section 2.7 we present the second part of our evaluation of FlowSpec through a data-flow analysis for the Stratego term transformation language. This section includes a comparative performance evaluation with the same data-flow analysis as currently implemented in the Stratego compiler. In Section 2.8 we compare against related work, and we conclude with Section 2.9.

2.2 Background: Data-flow Analysis and Monotone Frameworks

In this section we introduce data-flow analysis in general and monotone frameworks as a mathematical framework for sound, terminating data-flowanalysis.

2.2.1 Data-flow Analysis by Example

We start this introduction to data-flow analysis with two examples. Consider the *live variables* analysis in Figure 2.1. Live variables analysis provides the set of variable names, where the value currently bound to that variable *may* be read further along in the program. The figure shows an example program,

```
1   x = 2;
2   y = 4;
3   x = 1;
4   if(y > 0) {
5     z = y;
_   } else {
6     z = x * y;
_   }
7   x = z;
```

	LV$_\circ$	LV$_\bullet$
1	$\emptyset$	$\emptyset$
2	$\emptyset$	{y}
3	{y}	{x,y}
4	{x,y}	{x,y}
5	{y}	{z}
6	{x,y}	{z}
7	{z}	$\emptyset$

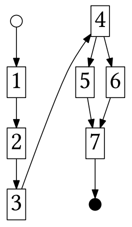


Figure 2.1 An illustration of *live variables* (LV) analysis. On the left is an example program in the While language, with numbered program fragments. On the right is the control flow graph (CFG) of the program. In the center is the analysis result. The LV$_\circ$ and LV$_\bullet$ are respectively before and after the variable accesses of the CFG node.

the results of live variables analysis for each statement, both before and after the effect of the statement, and the control-flow graph of the program. The control-flow graph shows how the program will execute either statement 5 or statement 6 based on whether condition 4 holds. Note how the variable `x` is only read in one branch of the `if` statement. Before the `if` statement, in the LV$_\circ$ set, `x` is still in the set as the analysis approximates the behaviour of both branches.

When a variable is not in the set of live variables after the statement that assigns a value to that variable, that means that the value assigned is not actually read. This information can be used by an optimisation to safely remove that assignment from the program. In the example, the assignment to `x` in statement 1 is such a redundant assignment, which can be recognised by the absence of `x` in the LV$_\bullet$ set of that first statement. This is consistent with the program, which does not read `x` until statement 6, and yet the variable is unconditionally reassigned in statement 3.

For comparison, we now discuss another data-flow analysis, *available expressions*, shown in Figure 2.2. Available expressions analysis provides the set of expressions that have already been computed. Expressions become unavailable again when a variable used in the expression is assigned a new value. Note that for an expression to be available, it needs to be available in *all* paths. At the start of the `while` loop, we can only consider expressions available when they are available right before the loop *and* at the end of the body of the loop. Therefore `a*b` is not available in AE$_\circ$ of condition 3, whereas `a+b` is.

The information from available expressions analysis can be used to remove redundant recomputations of expressions. We can save a repeated expression in a separate variable and use that variable instead of the expression, which is known as *common subexpression elimination*. In our example this would be expression `a+b` which can be replaced by `x` in condition 3.

In general, we note that for live variables analysis we need to know the behaviour in the next part of the program, whereas for available expressions

```
1   x = a + b;
2   y = a * b;
3   while(y > a+b) {
4     a = a + 1;
5     x = a + b
_   }
```

	$AE_\circ$	$AE_\bullet$
1	$\emptyset$	{a+b}
2	{a+b}	{a+b, a*b}
3	{a+b}	{a+b}
4	{a+b}	$\emptyset$
5	$\emptyset$	{a+b}

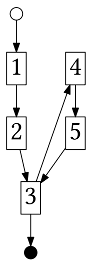


Figure 2.2 An illustration of *available expressions* (AE) analysis. On the left is an example program in the While language, with added brackets to number program fragments. On the right is the control flow graph (CFG) of the program. In the center is the analysis result. The open and closed dots on the analysis abbreviation are before and after a CFG node's effect respectively.

we need to know the expressions computed earlier. Therefore the computation of an analysis may need to propagate information either forward or backward. We also need to approximate the behaviour of the program when there are multiple paths to a program point. In live variables analysis we consider variables read in any path, whereas in available expressions we consider only expressions computed in all paths. It is useful to model these paths of control-flow with a control-flow graph, to abstract from concrete language constructs.

2.2.2 *Taxonomy of Data-flow Analysis*

The example analyses are both *flow-sensitive* analyses. These analyses take the control-flow of the program into account, i.e. the order in which effects occur. *Flow-insensitive* analysis is less accurate, but also computationally cheaper. This can be a useful trade-off for whole-program analysis, where procedure calls are taken into account. A refined form of flow sensitivity is the *path-sensitive* data-flow analysis, which derives information from conditionals as it takes one path or another. Information from conditionals is also used in the type systems of some programming languages (Pearce and Noble 2011; Jetbrains 2018; Red Hat, Inc. 2018), where the terminology is flow-sensitive types. In programming languages these flow-sensitive types are primarily used for conveniences such as tracking null-safety of pointer types, as well as more general structural sub-typing support.

The data-flow analyses we just presented are *intra-procedural*, i.e. they only consider code within procedures, and not procedure calls. By contrast, *inter-procedural* analysis takes calls into account. Since procedures can be called from multiple places, a sound analysis must either approximate over all contexts in which a procedure can be called, or the analysis must be *context-sensitive*. Different forms of context sensitivity exist. For example, *call-site sensitivity* is a form of context sensitivity that keeps a string of calls through which the current procedure was reached. A well-known *control-flow analysis*, is *k*-CFA (Shivers

1988).

When a programming language allows dynamic dispatch, e.g. through function pointers, higher-order functions, or inheritance, the control-flow from a call site can depend on run-time values. At this point inter-procedural data-flow analysis becomes interdependent with a dynamic control-flow analysis. This interaction between control-flow and data-flow is difficult to handle, as more approximation in one analysis, which speeds up that analysis, will result in more work for the other analysis. In object-oriented languages, the context sensitivity that shows potential for these analyses is *object-sensitivity*, which tracks the allocation sites of objects (Smaragdakis, Bravenboer and Lhoták 2011). Generally, finding the right contexts with a good trade-off in efficiency and accuracy are a topic of active research.

We aim to support flow-sensitive, intra-procedural data-flow analysis in FlowSpec as a start, which provides language designers with the tools to accurately analyse local properties.

2.2.3 Monotone Frameworks

Monotone frameworks (Kam and Ullman 1977) is a formal method for describing data-flow analyses. We give a short introduction to the framework here. Throughout this chapter we use the notation from F. Nielson, H. R. Nielson and Hankin (2005), which is the dual notation of the original publication (e.g. $\sqcup$ instead of $\wedge$).

In short, monotone frameworks is a general lattice theoretic framework for the definition of data-flow analyses. It captures the commonalities of intra-procedural, flow-sensitive data-flow analyses, and requires a number of components to be plugged in for any specific analysis. Given the correct components, this framework not only gives a clear, terminating semantics to a data-flow analysis, but also a simple worklist algorithm that can perform the analysis. The components required to instantiate a monotone framework are:

- The control-flow graph of a program in the form of a label set, an edge list of label pairs, and the starting labels.
- The type of data gathered by the data-flow analysis, along with its complete lattice instance of finite height. The framework uses lattice theory to guarantee a sound and terminating semantics.
- Transfer functions for every label in the control-flow graph, where the functions are monotone increasing with respect to the lattice.
- An initial value for the data-flow analysis at the starting labels.

2.2.4 Control-flow Graphs

In order to make a data-flow analysis flow-sensitive, we need the control-flow of a program. In monotone frameworks program fragments are labelled ($\ell \in \mathbf{Lab}$) to distinguish different parts of the program. A control-flow graph F is a set of edges (a subset of $\mathbf{Lab} \times \mathbf{Lab}$) between different labels.

For a forward propagating data-flow analysis, this graph can be used as is.

$$
\begin{aligned}
L &= \mathcal{P}(\mathbf{AExp}) \\
\sqsubseteq &= \supseteq \\
\sqcup &= \cap \\
\bot &= \mathbf{AExp}(\mathbf{Prog}) \\
\iota &= \emptyset \\
E &= \{\mathit{init}(\mathbf{Prog})\} \\
F &= \mathit{flow}(\mathbf{Prog})
\end{aligned}
$$

$$
\begin{aligned}
f_\ell(l) &= (l \setminus \mathit{kill}([B]^\ell)) \cup \mathit{gen}([B]^\ell) \\
&\qquad \text{where } [B]^\ell \in \mathit{blocks}(\mathbf{Prog})
\end{aligned}
$$

$$
\begin{aligned}
\mathit{kill}([x := a]^\ell) &= \{a' \in \mathbf{AExp}(\mathbf{Prog}) \mid x \in \mathit{FV}(a')\} \\
\mathit{gen}([x := a]^\ell) &= \{a' \in \mathbf{AExp}(a) \mid x \notin \mathit{FV}(a')\} \\
\mathit{gen}([b]^\ell) &= \mathbf{AExp}(b)
\end{aligned}
$$

Figure 2.3 The monotone framework instance for available expressions. An $l \in L$ is an element of the lattice. Two of those elements can be compared with $\sqsubseteq$, and joined with $\sqcup$. $\bot$ is the bottom of the lattice. The framework operates on the forward control-flow F, from the set of labels E, where the initial analysis information is ι. **Prog** is the entire program, *blocks* collects all labelled blocks, *FV* collects all free variables, *init* gives the initial label, and *flow* gives the control flow of the argument. **AExp** gives all arithmetic expressions.

For a backward propagating analysis the edges of the graph are simply flipped. The framework also takes the 'extremal labels' $E \in \mathcal{P}(\mathbf{Lab})$, which are the start nodes of the analysis.

2.2.5 Data-flow Type and Transfer Functions

Each control-flow graph node has an effect. An analysis specifies how this effect influences the analysis through a transfer function $f_\ell \colon L \to L$, where L is the type of the information the data-flow analysis propagates. At the extremal labels, this information is initialised with the extremal value ι.

An established factorisation of these transfer functions is the *kill* and *gen* sets approach. For set based analyses, a kill set is defined separately from a gen set for each control-flow node of interest. The transfer function is then generic: first remove the kill set, then add the gen set.

Figure 2.3 shows the monotone framework instance for available expressions analysis. The transfer function takes the set of available expressions, first removes any expressions containing the variable that is assigned (or nothing if it is not an assignment), then it adds any new expressions that do not contain the variable that is assigned. Note how the gen set has to repeat the conditions of the kill set. The reason for this repetition is that the right-hand side of the expression, that generates available expressions, happens before the assignment effect of the left-hand side. In a backward analysis this would not be the case, therefore on first glance independent kill and gen sets are *sometimes* dependent. We argue that this subtlety can be a source of analysis bugs.

2.2.6 Control-flow and Lattices

If a control-flow graph node ℓ has the information $\text{Analysis}_\circ(\ell)$ before the effect of ℓ then we can use its transfer function f_ℓ to compute $\text{Analysis}_\bullet(\ell)$. However, when multiple control-flow paths join at a certain node, we need to merge the data from those different paths. We use the $\sqcup$ operator for this to reach these equations:

$$\begin{aligned}
\text{Analysis}_\circ(\ell) &= \bigsqcup\{\text{Analysis}_\bullet(\ell') \mid (\ell',\ell) \in F\} \sqcup \iota_E^\ell \\
&\quad \text{where } \iota_E^\ell = \begin{cases} \iota & \text{if } \ell \in E \\ \bot & \text{if } \ell \notin E \end{cases} \\
\text{Analysis}_\bullet(\ell) &= f_\ell(\text{Analysis}_\circ(\ell))
\end{aligned}$$

The open dot is the analysis result before the effect of ℓ, and the closed dot is for after the effect of ℓ. The transfer function f_ℓ is used to compute the effect of ℓ. The $\sqcup$ operator is used to combine the analysis data after the previous nodes ℓ' as the analysis data right before the current node ℓ. We use the initial value ι for the initial labels E and a $\bot$ value elsewhere, where $\bot \sqcup d = d = d \sqcup \bot$.

Finding the fixed point to these equations may not be possible though, as loops in the control-flow graph make the equations recursive. Therefore we need stronger guarantees, for which lattices are used.

Monotone frameworks require a complete lattice instance $(\top, \bot, \sqsubseteq, \sqcup, \sqcap)$ for the type L of the data-flow property. The intuition is that $\top$ is the value of L that reads as "could be anything", the coarsest approximation available. By using the least upper-bound operator ($\sqcup$) we combine the information from two paths in the control-flow so it soundly approximates both (upper bound), while keeping as much information as possible (*least* upper bound).

In Figure 2.3, the monotone frameworks instance of available expressions uses a powerset lattice. Available expressions analysis is a must analysis, which only keeps information that *must* be true for *all* paths. Therefore the analysis applies set intersection at join-points.

Now that we have a clearer definition of the $\sqcup$ operator, we can resolve the issue of finding a fixed point to the equations. Monotone frameworks have two particular requirements. First, the transfer functions f_ℓ need to be monotone increasing with respect to the lattice. This means that in a loop either the information becomes more approximate, or it stays the same, in which case we have a fixed point. Secondly, the lattice must adhere to the ascending chain condition. In other words, the lattice must have a finite height. This way when the information on a loop keeps increasing, it takes a finite number of steps to reach $\top$, which is a fixed point for monotone increasing transfer functions.

Of course $\top$ is the coarsest approximation available. Although some approximation is necessary to keep the analysis computable, we can usually do better than $\top$ everywhere. The fixed point of the Analysis that we want is the *least fixed point*. This fixed point has enough information to be valid, with as little approximation as necessary. The accuracy of this fixed point is still dependent on the choice of lattice L and transfer functions f.

In the original work on monotone frameworks (Kam and Ullman 1977) the dual notion with meets (greatest lower bounds) and greatest fixed points was used. There, the authors give the Meet Over all Paths (MOP) as the

```
for unique $\ell$ in $F$:
  if $\ell \in E$:
    Analysis$_\circ$($\ell$) := $\iota$
  else:
    Analysis$_\circ$($\ell$) := $\bot$

$W$ := list-of($\mathbf{Lab}$)

while $W \neq []$:
  $\ell$ := $W$.pop()
  for ($\ell$, $\ell'$) $\in F$:
    if $f_\ell$(Analysis$_\circ$($\ell$)) $\not\sqsubseteq$ Analysis$_\circ$($\ell'$):
      Analysis$_\circ$($\ell'$) := $f_\ell$(Analysis$_\circ$($\ell$)) $\sqcup$ Analysis$_\circ$($\ell'$)
      $W$.push($\ell'$)

for unique $\ell$ in $F$:
  Analysis$_\bullet$($\ell$) := $f_\ell$(Analysis$_\circ$($\ell$))
```

Listing 2.1 A worklist algorithm to iteratively solve the equations of a monotone framework instance. W is the worklist. **Lab** is the set of labels used in the control-flow graph F.

desired solution, but show that this solution can be undecidable to calculate. In cases where it *can* be calculated, the greatest fixed point coincides with it, in cases where it is undecidable, the greatest fixed point safely approximates the MOP solution (F. Nielson, H. R. Nielson and Hankin 2005, § 2.4.2). Therefore a correct instantiation of a monotone framework gives a computable, safe approximation of the run-time behaviour of a program.

2.2.7 Worklist Algorithm

Given an instance of a monotone framework, we can compute the fixed point of the recursive equations iteratively with a worklist algorithm, such as the one in Listing 2.1. This algorithm works in three steps. First it initialises the analysis result `Analysis`$_\circ$ to what comes down to ι_E^ℓ, and the worklist to all nodes in the control-flow graph. Second, it loops over the worklist, taking out one node at a time, and propagates transferred analysis information to successors in the control-flow graph. If that information is new ($\not\sqsubseteq$) the $\sqcup$ operator is used to add the information to the analysis information of the successor, and that successor is added to the worklist again. Once no more new information is discovered, the worklist becomes empty. The third step computes `Analysis`$_\bullet$ as defined in its formula.

2.2.8 Monotone Frameworks Recap

To summarise, to specify a data-flow analysis with monotone frameworks, we need the following ingredients:

1. A finite flow, $F \in \mathcal{P}(\mathbf{Lab} \times \mathbf{Lab})$.
2. Labels $\ell \in \mathbf{Lab}$, which reference program fragments.
3. A set of extremal labels, $E \in \mathcal{P}(\mathbf{Lab})$, typically the initial label(s) of the

flow.

4. A type L of the data-flow property, which is a complete lattice of finite height.
5. Monotone transfer functions f_ℓ for every label ℓ in the control-flow graph.
6. An extremal value, $\iota \in L$, for the extremal labels.

Monotone frameworks give a design pattern for correct data-flow analysis, and an implementation for such an analysis. However, direct instantiations of worklists for different analyses, especially analyses that use the results of other analyses can result in a complex implementation that is difficult to update or adapt.

In language workbenches we want to specify a data-flow analysis and get the implementation 'for free', i.e. we abstract from the implementation method. The iterative algorithm can still be used under the hood, but is no longer directly seen. The specification should be easy to understand and avoid pitfalls such as we saw in the gen and kill set definition of available expressions. In short, we need a domain-specific language for data-flow analysis specification.

2.3 FlowSpec by Example

FlowSpec is a domain-specific language for specifying data-flow analysis, that builds on the theory of monotone frameworks. A FlowSpec specification only includes the analysis-specific elements, and from this specification we generate an implementation for that analysis. In this section we introduce FlowSpec by a number of examples.

2.3.1 Requirements

In language workbenches we want to specify a data-flow analysis and get the implementation 'for free', i.e. we abstract from the implementation method. The specification should be easy to understand and avoid pitfalls such as we saw in the gen and kill set definition of available expressions for monotone frameworks.

Within the context of a language workbench, we need a language that reuses information that is already available within a language specification of the workbench. We do not need to define a data-flow analysis directly on source text of a program, as we can obtain the abstract syntax of that program within the workbench. We can also reuse name and type analysis that is available. Our domain-specific language does not need to support the specification of such analyses, it should only support the use of the analysis results.

What we need then is a language that uses the concepts of abstract syntax, names and types, and provides features to define what is distinctly part of the domain of data-flow analysis. FlowSpec provides the features to define the relation between the control-flow and the abstract syntax of a programming language, and what effects control-flow nodes of the programming language have for different data-flow analyses.

```
module running-example

context-free syntax

  Statement.Seq =
    [[Statement]
     [Statement]] {right}

  Statement.Assign = [[ID] = [Expr];]

  Statement.IfThenElse =
    [if([Expr]) [Statement]
     else [Statement]]

  Statement.While =
    [while([Expr])
       [Statement]]

lexical syntax

  ID  = [a-zA-Z] [a-zA-Z0-9\_]*
  INT = "-"? [0-9]+


context-free priorities

  { Expr.UMin  Expr.Not } >
  { left: Expr.Mul  Expr.Div  Expr.Mod } >
  { left: Expr.Add  Expr.Sub } >
  { left: Expr.Lt  Expr.Leq
          Expr.Gt  Expr.Geq } >
  { left: Expr.Eq  Expr.Neq } >
  Expr.And > Expr.Or
```

```
context-free syntax

  Expr.IntLit = INT
  Expr.True   = [true]
  Expr.False  = [false]
  Expr.VarRef = ID

  Expr     = [([Expr])] {bracket}

  Expr.UMin = [-[Expr]]
  Expr.Mul  = [[Expr] * [Expr]] {left}
  Expr.Div  = [[Expr] / [Expr]] {left}
  Expr.Add  = [[Expr] + [Expr]] {left}
  Expr.Sub  = [[Expr] - [Expr]] {left}

  Expr.Not = [![Expr]]
  Expr.And = [[Expr] && [Expr]] {left}
  Expr.Or  = [[Expr] || [Expr]] {left}
  Expr.Eq  = [[Expr] == [Expr]] {left}
  Expr.Gt  = [[Expr] > [Expr]]  {left}
  Expr.Lt  = [[Expr] < [Expr]]  {left}
  Expr.Geq = [[Expr] >= [Expr]] {left}
  Expr.Leq = [[Expr] <= [Expr]] {left}
  Expr.Neq = [[Expr] != [Expr]] {left}
```

Listing 2.2 The SDF3 grammar for the running example language While.

2.3.2 *Concrete and Abstract Syntax*

In Spoofax the concrete and abstract syntax of a programming language are defined in SDF3. As an example, we provide the SDF3 definition of the syntax of our running example language in Listing 2.2.

This SDF3 grammar uses templates to specify grammar rules along with some basic formatting hints. Within the outer brackets (either square or angled), are terminals, within another pair of brackets are non-terminals. The first rule defines that a statement can be a sequence of two statements. The annotation `right` disambiguates this rule by making the rule right-associative. In other words, a sequence of three statements S_1 S_2 S_3 is parsed as S_1 $(S_2$ $S_3)$. In this example grammar we define statements as sequences of statements, assignment statements, if statements and while loop statements.

We define expressions within conditions and assignment right-hand sides. Expressions include a number of binary and unary operations which have associativity and priority to disambiguate. The lexical syntax for identifiers and integer literals is defined at the end with some regular expressions.

The abstract syntax of our running example is already written as part of the

```
signature
  constructors // generated from SDF3 spec
    Seq : Statement * Statement -> Statement
    Assign : ID * Expr -> Statement
    IfThenElse :
      Expr * Statement * Statement -> Statement
    While : Expr * Statement -> Statement
  constructors
    VarRef : ID -> Expr
  constructors // manually defined to desugar into
    BinOp : BinOp * Expr * Expr -> Expr
    UnOp  : UnOp * Expr -> Expr
  constructors                                constructors
    UMin : UnOp                                 Or   : BinOp
    Mul  : BinOp                                Eq   : BinOp
    Div  : BinOp                                Gt   : BinOp
    Add  : BinOp                                Lt   : BinOp
    Sub  : BinOp                                Geq  : BinOp
    Not  : UnOp                                 Leq  : BinOp
    And  : BinOp                                Neq  : BinOp
```

Listing 2.3 The abstract syntax of the running example language While.

```
x = a + b;            Seq(Assign("x", BinOp(Add(), VarRef("a"), VarRef("b"))),
y = a * b;            Seq(Assign("y", BinOp(Mul(), VarRef("a"), VarRef("b"))),
while(y > a + b) {    While(BinOp(Gt(), VarRef("y"), BinOp(Add(), ...)),
  a = a + 1;          Seq(Assign("a", BinOp(Add(), VarRef("a"), IntLit("1"))),
  x = a + b;          Assign("x", BinOp(Add(), VarRef("a"), VarRef("b")))
}                     ))))
```

Listing 2.4 An example program with its desugared abstract syntax tree.

SDF3 grammar, in the form of constructor names on the rules. A sequence of statements uses `Seq`, an addition uses `Add`, et cetera. From the grammar we can generate the signatures of the abstract syntax, as shown in Listing 2.3. The first two sections of signatures define the shape of the abstract syntax tree (AST) as defined in the grammar. However, we have *desugared* unary and binary operations to common constructors that have a separate operator field. These constructor signatures are hand-written, and some simple transformation rules can translate the original AST to this desugared version that we will operate on throughout the examples.

In Listing 2.4 we give an example program along with its abstract syntax tree. The different Spoofax meta-languages, including FlowSpec, work with these ASTs by pattern matching against parts of the tree.

2.3.3 *Name and Type Analysis*

Name and type analysis is extracted from an NaBL2 specification. This analysis annotates the entire tree with unique numbers, so different *occurrences* of a name can be distinguished from each other. All information of names and types is then attached to these occurrences.

The reason we care about name and type analysis, is that data-flow analysis commonly gathers information about names. Many programming languages allow shadowing of names, i.e. definition of a name in an inner scope when the same name is present already in an outer scope. Therefore the name of a variable is not unique enough when data-flow analysis collects information on that name.

In FlowSpec we operate on analysed ASTs, in which name occurrences have unique numbers. Within FlowSpec we borrow the NaBL2 notation `Namespace{name}` for name occurrences. In FlowSpec, such an occurrence denotes names *after* name analysis. That is, names represented by the same occurrence, correspond to the same declaration. Thus, name capture is not a concern in FlowSpec.

2.3.4 Control-flow Graphs

For flow-sensitive data-flow analysis we require a control-flow graph. Control-flow graphs are a finite representation of a possible infinite set of paths through a program. For example, as statements of a program are executed, control flows from one statement to the next. However, as soon as a program has a loop, control can flow around the loop or at some point exit the loop. A control-flow graph is a finite model that show where control *could* flow. Thereby a loop in a program become in a loop in a control-flow graph. Examples of control-flow graphs can be found back in Figure 2.1 on page 19 and Figure 2.2 on page 20.

2.3.5 Mapping from Abstract Syntax to Control Flow

In FlowSpec we build control-flow graphs between occurrences in the AST. Not only a string occurrence as is usually used for names, but also entire subtrees of the AST can be control-flow graph nodes. The FlowSpec specification defines which AST nodes should be considered control-flow graph nodes, and how control flows within and between different AST nodes.

Consider Listing 2.5 where we have defined some example mapping rules. The rules are defined case-by-case using patterns to match the signature from Listing 2.3. Each rule uses the contextual `entry` and `exit` nodes to connect the sub-graph of the matched AST node to the outer graph. These nodes do not show up in the final control-flow graph. When the AST node matched by the pattern should be included as a control-flow node, we use the `this` keyword to denote that. The direct use of a pattern variable from the AST pattern is substituted with the subgraph of that AST node in accordance with other control-flow rules.

Listing 2.6 shows how control-flow rules can be applied to a program in a number of steps. First the sequence rule is used to create an edge between the two statements. Then the assignment rule is used to create nodes of the assignments, with the expression preceding it. Then the binary operation rule creates a node for the operation and its operands, and finally the operands are turned into nodes themselves.

The rule of `IfThenElse` shows that multiple chains of edges in the control-flow graph may be defined. This allows us to express conditions. The sub-graph of condition `c` is followed by both the subgraph of the then-branch and subgraph of the else-branch. Each of the branches connect to the `exit` of the construct.

```
module control

control-flow rules
  Assign(_, e) = entry -> e -> this -> exit

  Seq(s1, s2) = entry -> s1 -> s2 -> exit

  IfThenElse(c, t, e) = entry -> c -> t -> exit,
                                  c -> e -> exit

  While(c, b) = entry -> c -> b -> c -> exit

  BinOp(_, l, r) = entry -> l -> r -> this -> exit

  UnOp(_, e) = entry -> e -> this -> exit

  node VarRef(_)
  node IntLit(_)
  node True()
  node False()
```

Listing 2.5 Control-flow graph rules for the While language. Each rule can have one or more chains of edges, where `entry` and `exit` represent the connection between the local control flow the rest of the graph.

```
⟦Seq(
  Assign("a", BinOp(Add(), VarRef("a"), IntLit("1"))),
  Assign("x", BinOp(Add(), VarRef("a"), VarRef("b")))
)⟧

entry ->
⟦Assign("a", BinOp(Add(), VarRef("a"), IntLit("1")))⟧ ->
⟦Assign("x", BinOp(Add(), VarRef("a"), VarRef("b")))⟧ ->
exit

entry ->
  ⟦BinOp(Add(), VarRef("a"), IntLit("1")))⟧ -> node Assign("a", ...) ->
  ⟦BinOp(Add(), VarRef("a"), VarRef("b")))⟧ -> node Assign("x", ...) ->
exit

entry ->
  ⟦VarRef("a")⟧ -> ⟦IntLit("1")⟧ ->
  node BinOp(Add(), ..., ...)) -> node Assign("a", ...) ->
  ⟦VarRef("a")⟧ -> ⟦VarRef("b")⟧ ->
  node BinOp(Add(), ..., ...)) -> node Assign("x", ...) ->
exit

entry ->
  node VarRef("a") -> node IntLit("1") ->
  node BinOp(Add(), ..., ...)) -> node Assign("a", ...) ->
  node VarRef("a") -> node VarRef("b") ->
  node BinOp(Add(), ..., ...)) -> node Assign("x", ...) ->
exit
```

Listing 2.6 Control-flow graph rules applied to a piece of abstract syntax, where double square brackets show parts of the AST that are not processed yet.

```
module liveness

properties
  live : MaySet(name)

property rules
  live(Assign(n, _) -> s) = live(s) \ { Var{n} }

  live(VarRef(n) -> s) = live(s) \/ { Var{n} }

  live(_ -> s) = live(s)
```

Listing 2.7 Live Variables specification in FlowSpec. Note how the rule for assignments does not inspect the right-hand side expression. Instead the control-flow is defined within expressions (not in this figure), and a separate rule for the variable reference expressions adds live variables. Names are added to the live variables as names within a namespace `Var`. External name information is used to handle name issues such as shadowing.

Multiple uses of the `c` sub-graph refer to the same subgraph. This is also used in the `While` rule, where multiple uses of condition `c` construct the loop.

In these rules binary and unary operations, i.e. expressions, are also considered part of the control-flow graph. This is not a restriction in FlowSpec, we define the control-flow this way to the benefit of our data-flow analysis definitions later. One could also use `node c` within the chain of edges to make condition expression `c` a node in the control-flow graph.

The rules for variable references and integer literals are a shorthand to define that this is a node in the control-flow graph and it has no further control-flow inside. The following would be equivalent to the variable reference rule:

```
VarRef(_) = entry -> this -> exit
```

2.3.6 Data-Flow Type and Transfer Functions

FlowSpec defines data-flow analyses as properties on the control-flow graph. During analysis, the data of this property is propagated along the control-flow graph. Every node in the control-flow graph has an associated effect on this data.

In Listing 2.7 we show FlowSpec's analogue of transfer functions for live variables analysis: property rules. Property rules show both the direction of the data-flow analysis, in this case backward, and define the data-flow property in terms of itself. We have a rule for assignments, which only applies the effect of the assignment itself and disregards the right-hand side expression. A separate rule for the variable reference expression handles the effect of reading a variable. We are able to split these two effects because earlier we defined control-flow in expressions too.

The FlowSpec specification of available expressions is given in Listing 2.8. We use an external property `refs` from NaBL2 to extract references from expressions. The effects of the assignment and its right-hand side expression

```
module availability

properties
  available: MustSet(term)
  external
    refs: Set(name)
property rules
  available(prev -> Assign(n, _)) =
    { expr |
      expr <- available(prev),
      !(Var{n} in refs(expr) }

  available(prev -> e@BinOp(_,_,_)) =
    available(prev) \/ {e}

  available(prev -> e@UnOp(_,_,_)) =
    available(prev) \/ {e}

  available(prev -> _) = available(prev)
```

Listing 2.8 Available Expressions specification in FlowSpec. We consider references in expressions a separate concern based on names, not flow and therefore out of scope for our language. Note how the assignment rule only handles the assignment effect, expressions are visited separately.

are split over multiple rules again. The assignment filters out those expressions that use the variable that is being assigned to. We can express this as a direct filter instead of relying on global program information of all expressions, as was the case in the monotone frameworks definition in Figure 2.3 on page 22. Since our control-flow graph includes the expressions in an assignment separately, it models the ordering of effects directly. Therefore the FlowSpec specification does not suffer from the subtle interdependence that the traditional kill-gen definitions have.

2.3.7 Lattices and Termination

The control-flow can split and join because of conditional control-flow such as an `if` statement. We can propagate data along both edges of a split, but need to merge the data coming from multiple directions at a join. The data is merged before the property rule of the join-point node is applied. We require a lattice instance $(\top, \bot, \sqsubseteq, \sqcup, \sqcap)$ for the type of the data-flow property, and use the least-upper bound $\sqcup$ at join points in the control-flow. In our examples the `MaySet` and `MustSet` are lattice instances that use the `Set` type:

```
properties
  live : MaySet(name)
  available : MustSet(term)
```

A `MaySet` performs set unions at control-flow join points and compares with non-strict subset comparison. A `MustSet` uses intersection and non-strict superset comparison. It uses a symbolic bottom element to represent the full set of possible values in the analysis.

```
module busyness

properties
  external
    refs: Set(name)

properties
  veryBusy: MustSet(term)

property rules
  veryBusy(Assign(n, e) -> next) =
    { expr |
      expr <- veryBusy(next),
      !(n in refs(expr)) }

  veryBusy(e@BinOp(_,_,_) -> next) =
    veryBusy(next) \/ {e}

  veryBusy(e@UnOp(_,_,_) -> next) =
    veryBusy(next) \/ {e}

  veryBusy(_ -> next) = veryBusy(next)
```

Listing 2.9 Very Busy Expressions specification in FlowSpec.

```
module reach

properties
  definition: MaySet(name * occurrence)

property rules

  definition(prev -> this@Assign(n, e)) =
    { (Var{n}, occurrence(this)) } \/
    { (m, l) |
      (m, l) <- definition(prev),
      m != Var{n} }

  definition(prev -> _) = definition(prev)
```

Listing 2.10 Reaching Definitions specification in FlowSpec.

2.3.8 *Very Busy Expressions*

Very busy expression analysis provides the set of expressions which will definitely be calculated in the future. This information can be used to hoist an expression out of an if statement if it is calculated in both branches. In Listing 2.9 we provide the definition of very busy expressions analysis in FlowSpec. Note how similar this analysis is to the available expressions analysis.

2.3.9 *Reaching Definitions*

Reaching definition analysis is an analysis that provides the positions in the program where a variable was last assigned a value. This can be multiple positions since a variable may be assigned in different conditional paths in

```
module constants

prop constProp: CP

constProp(prev -> Assign(n, e)) =
  match constProp(prev) with
    | M1(m, v) => M(m \/ {Var{n} |-> v})
    | _ => CP.top

constProp(prev -> Add(e1,e2)) =
  match constProp(prev) with
   | M2(m, l, r) => M1(m, constAdd(l,r))
   | _ => CP.top

constProp(prev -> VarRef(n)) =
  addResult(
    constProp(prev),
    getMap(constProp(prev))[Var{n}])

constProp(prev -> _) = constProp(prev)
```

Listing 2.11 Constant propagation specification in FlowSpec. `CP` holds the map of names to constants and 0, 1 or 2 constants from previous computations.

the control-flow. See Listing 2.10 for the FlowSpec description of reaching definitions. To preserve the information from all branches, we use a `MaySet`. The `occurrence` is used to denote the position in the program where the assignment occurred.

The sole rule for reaching definitions analysis of our example language is that of the assignment. There we remove any previously reaching definitions of the currently assigned variable. We add the pair of the name and the occurrence of the assignment.

2.3.10 Constant Propagation and Folding

Constant propagation is the name of both an analysis and the corresponding optimisation. The optimisation replaces uses of a variable with its value if that value is guaranteed to be constant. Constant folding is the optimisation that computes constant expressions and replaces those expressions with the computed result. We combine these two optimisations and make them part of our constant propagation, to improve the accuracy of the analysis results. Because of constant folding, more constants can be found. Because more constants can be found, and filled into expressions, more constant expressions can be folded.

In Listing 2.11 and Listing 2.12 we give the definition of this combined constant propagation analysis in FlowSpec. The constant propagation property had a `Map` type. A FlowSpec `Map` forms a lattice if the value type forms a lattice. Any key not bound in the map, instead maps to the top of the lattice of the value type. The $\sqcup$ and $\sqsubseteq$ operators are defined point-wise. This means that if a variable is only constant in one condition branch, when it joins with another

```
types
  CPType =
    | M(Map(name, Const))
    | M1(Map(name, Const), ConstProp)
    | M2(Map(name, Const), ConstProp, ConstProp)

  ConstProp =
    | Top()
    | Const(int)
    | Bottom()

functions
  getMap(cpt: CPType) =
    match cpt with
      | M(m) => m
      | M1(m,_) => m
      | M2(m,_,_) => m

  addResult(cpt: CPType, v: Const) =
    match cpt with
      | M1(m, v1) => M2(m, v1, v)
      | _ => M1(getMap(cpt), v)

  constAdd(l: Const, r: Const) =
    match(l,r) with
      | (Const(i), Const(j)) => Const(i+j)
      | _ => Const.top

lattices
  CP where
    type CPType

    lub(l, r) = match (l,r) with
      | (M(l), M(r)) => M(Map.lub(l,r))
      | (M1(l, cl), M1(r, cr)) => M1(Map.lub(l,r), Const.lub(cl,cr))
      | (M2(l, cl1, cl2), M2(r, cr1, cr2)) => M2(Map.lub(l,r), Const.lub(cl1,
  cr1), Const.lub(cl2, cr2))
      | _ => CP.top

    bottom = M(Map.bottom)

    top = M(Map.top)

  Const where
    type = ConstProp

    lub(l, r) = match (l,r) with
      | (Top(), _) => Top()
      | (_, Top()) => Top()
      | (Const(i), Const(j)) => if i == j then Const(i) else Top()
      | (_, Bottom()) => l
      | (Bottom(), _) => r

    bottom = Bottom()
```

Listing 2.12 The type, function and lattice definitions for *constant propagation* specification in FlowSpec.

```
module sign

properties
  sign : MaySet(name * Sign)

property rules
  sign(prev -> Assign(n, e)) =
    (Var{n}, interpSet(sign(prev), e)) \/
    { (m, s) | (m, s) <- sign(prev), m != Var{n} }

  sign(prev -> _) = sign(prev)

functions
  interpSet(set, e) = match e with
    | VarRef(n) => { s | (m, s) <- set, m == Var{n} }
    | BinOp(op,e1,e2) =>
      { s | l <- interpSet(set, e1),
            r <- interpSet(set, e2),
            s <- interpBin(op,l,r) }
    // etc.

  interpBin(op, l, r) = match op with
    | Add() => (if l == r
      then { l }
      else match (l, r) with
        | (_, Zero()) => { l }
        | (Zero(), _) => { r }
        | _ => { Pos(), Neg(), Zero() })
    // etc.

types
  Sign =
    | Zero()
    | Neg()
    | Pos()
```

Listing 2.13 Sign Analysis specification in FlowSpec.

branch the variable will no longer be considered constant.

The constant value lattice has a symbolic top and bottom, and constants which are not ordered. Therefore when two branches in the control-flow join, and different constant values for the same variable are found, that variable is no longer constant at the join point. The constant propagation property rule takes assignment into account and applies the `foldConst` function, which computes constant expressions.

2.3.11 Sign Analysis

Sign analysis is a data-flow analysis that computes the possible sign of integers typed variables. This can be used to detect if a comparison condition will always evaluate to a constant, which makes further analysis more accurate as a branch of control-flow is eliminated. Sign analysis is similar in definition to constant propagation as illustrated in Listing 2.13.

```
module initialization

properties
  definite: MustSet(name)

property rules

  definite(prev -> this@Assign(n, e)) =
    definite(prev) \/ { Var{n} }

  definite(prev -> _) = definite(prev)
```

Listing 2.14 Definite Assignment specification in FlowSpec.

2.3.12 *Definite Assignment*

Definite assignment analysis keeps a set of variables which have definitely been assigned a value. This information can be used to give warnings or error upon the use of a possibly uninitialised variable. The FlowSpec specification of definite assignment is in Listing 2.14.

2.4 The Semantics of FlowSpec

In this section we present the semantics of FlowSpec. For brevity we only show rules for the novel parts of the language, and use monotone frameworks as the semantic model for the language. We will discuss the language in roughly the same order as in the last section. Please refer to Figure 2.4 for a small syntax definition of the language, from which we will use non-terminals to introduce judgements of the semantics.

Note that the `this` construct in FlowSpec is syntactic sugar for a `node t` where `t` refers to the entire AST that was matched with pattern p.

2.4.1 *Control-flow Rules*

The control-flow rules, that map the abstract syntax of a language to its control-flow, are defined case-wise with AST patterns. To model the behaviour of the virtual entry and exit nodes in these rules, we employ a constraint based semantics, given in Figure 2.5 on page 38. The smallest set that satisfies these constraints is the control-flow graph that the rules define. We use $[\![p]\!]^{a^{\ell}} = \Gamma$ to abstract over pattern matching, where p is the pattern, a^{ℓ} is the labeled AST node, and Γ is the environment with bindings that come from the match. The extremal labels are all possible, valid bindings of $\ell_{\circ}$ and $\ell_{\bullet}$ for [rule$_i$] where a^{ℓ} is the whole program.

In general the four labels left of the turnstile are the virtual entry and exit labels, and the start and end labels. The entry and exit labels are mostly left to be inferred by the rules. The chain rule [noedge] connects the labels in a chain by using an inference variable as a label to connect the two judgements. The chain rule [edge] connects the labels by using two inference variables and adding an edge between these variables to the graph.

For the chain element rules [en] and [ex] we assume that entry nodes are

$S ::=$	$\texttt{control-flow rules}\ \overline{G}$	control-flow section
$\mid$	$\texttt{properties}\ \overline{D}$	dataflow prop def section
$\mid$	$\texttt{property rules}\ \overline{R}$	dataflow prop rules section
$G ::=$	$p\ \texttt{=}\ C\ \overline{\{, C\}}$	control-flow rules
$C ::=$	$E \to E\ \overline{\{\to E\}}$	edge chains
$E ::=$	$\texttt{entry} \mid \texttt{exit} \mid \texttt{start} \mid \texttt{end}$	chain elements
$\mid$	$n \mid \texttt{node}\ n$	
$D ::=$	$n\ \texttt{:}\ t$	property definitions
$R ::=$	$n\ P\ \texttt{=}\ e$	property rules
$P ::=$	$p \to n$	match ahead
$\mid$	$n \to p$	match behind
$p ::=$	$n\texttt{(}\overline{p}\texttt{)}$	term pattern
$\mid$	$\texttt{(}\overline{p}\texttt{)}$	tuple pattern
$\mid$	$\_$	wildcard pattern
$\mid$	n	pattern variable
$\mid$	$n\texttt{@}p$	as pattern
$e ::=$	n	variable reference
$\mid$	$n\texttt{(}n\texttt{)}$	property lookup
$\mid$	$n\texttt{(}\overline{e}\texttt{)}$	function application
$\mid$	$\texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e$	if else
$\mid$	$\texttt{match}\ e\ \texttt{with}\ \overline{\{\texttt{\lvert}\ p \Rightarrow e\}}$	pattern match
$\mid$	$\texttt{type(}n\texttt{)}$	type lookup
$\mid$	$n\ \texttt{\{}n\texttt{\}}$	name lookup
$\mid$	$\texttt{occurrence(}n\texttt{)}$	occurrence lookup
$\mid$	$e\ \texttt{==}\ e \mid e\ \texttt{!=}\ e \mid \texttt{!}e$	equality, inequality and negation
$\mid$	$\texttt{(}\overline{e}\texttt{)} \mid n\texttt{(}\overline{e}\texttt{)}$	tuple and term literals
$\mid$	$s \mid i$	string and number literals
$\mid$	$\texttt{\{}\overline{e}\texttt{\}}$	set literal
$\mid$	$\texttt{\{}e\ \texttt{\lvert}\ p \leftarrow e\overline{\{\texttt{,}p \leftarrow e\}\{\texttt{,}e\}}\texttt{\}}$	set comprehension
$\mid$	$\texttt{\{}\overline{\{e \mapsto e\}}\texttt{\}} \mid e\texttt{[}e\texttt{]}$	map literal and lookup
$\mid$	$e \cup e \mid e \setminus e \mid e\ \texttt{in}\ e \mid e \cap e$	set operations
n		names
t		types
s		string
i		number

Figure 2.4 The core grammar of FlowSpec.

only on the left-most end of a chain, and exit nodes are only on the right-most end of a chain. The entry and exit rules simply equate the two labels left of the turnstile, without putting any constraints on the two labels. The [end] and [start] rules are similar, except these use the downward propagated **start**$^\ell$ and **end**$^\ell$ nodes that are created by the `root` rule. The [lab] rule looks up the label of the AST node, and requires that both labels left of the turnstile are equal to this label. This rule is the one that adds an actual label to the system of rules, instead of an inference variable. This forces the [edge] rule to be used between

Cfg root rule constraints $\boxed{\vdash [\![G]\!]^{a^\ell} \supseteq \textbf{Lab} \times \textbf{Lab}}$

$$\frac{\begin{array}{c}[\![p]\!]^{a^\ell} = \Gamma \quad 1 \le i \le m \\ \textbf{start}^\ell, \textbf{end}^\ell, \textbf{start}^\ell, \textbf{end}^\ell; \Gamma \vdash C_i \supseteq g_i\end{array}}{\vdash [\![\textbf{root}\ p = C_1, \ldots, C_m]\!]^{a^\ell} \supseteq g_i}\ [\mathsf{rule}_i]$$

Cfg rule constraints $\boxed{\ell, \ell, \ell, \ell \vdash [\![G]\!]^{a^\ell} \supseteq \textbf{Lab} \times \textbf{Lab}}$

$$\frac{[\![p]\!]^{a^\ell} = \Gamma \quad \ell_\circ, \ell_\bullet, \ell_\mathbf{s}, \ell_\mathbf{e}; \Gamma \vdash C_i \supseteq g_i \quad 1 \le i \le m}{\ell_\circ, \ell_\bullet, \ell_\mathbf{s}, \ell_\mathbf{e} \vdash [\![p = C_1, \ldots, C_m]\!]^{a^\ell} \supseteq g_i}\ [\mathsf{rule}_i]$$

Cfg chain constraints $\boxed{\ell, \ell, \ell, \ell; \Gamma \vdash C \supseteq \textbf{Lab} \times \textbf{Lab}}$

$$\frac{\begin{array}{c}\ell_\circ, \ell, \ell_\mathbf{s}, \ell_\mathbf{e}; \Gamma \vdash E_1 \supseteq g_1 \\ \ell, \ell_\bullet, \ell_\mathbf{s}, \ell_\mathbf{e}; \Gamma \vdash E_2 \to \ldots \to E_m \supseteq g_2\end{array}}{\ell_\circ, \ell_\bullet, \ell_\mathbf{s}, \ell_\mathbf{e}; \Gamma \vdash E_1 \to E_2 \to \ldots \to E_m \supseteq g_1 \cup g_2}\ [\mathsf{noedge}]$$

$$\frac{\begin{array}{c}\ell_\circ, \ell_1, \ell_\mathbf{s}, \ell_\mathbf{e}; \Gamma \vdash E_1 \supseteq g_1 \quad \ell_1 \neq \ell_2 \\ \ell_2, \ell_\bullet, \ell_\mathbf{s}, \ell_\mathbf{e}; \Gamma \vdash E_2 \to \ldots \to E_m \supseteq g_2 \\ g_3 = \{(\ell_1, \ell_2)\}\end{array}}{\ell_\circ, \ell_\bullet, \ell_\mathbf{s}, \ell_\mathbf{e}; \Gamma \vdash E_1 \to E_2 \to \ldots \to E_m \supseteq g_1 \cup g_2 \cup g_3}\ [\mathsf{edge}]$$

Cfg element constraints $\boxed{\ell, \ell, \ell, \ell; \Gamma \vdash E \supseteq \textbf{Lab} \times \textbf{Lab}}$

$$\frac{}{\ell_\circ, \ell_\circ, \ell_\mathbf{s}, \ell_\mathbf{e}; \Gamma \vdash \textbf{entry} \supseteq \emptyset}\ [\mathsf{en}] \qquad \frac{}{\ell_\bullet, \ell_\bullet, \ell_\mathbf{s}, \ell_\mathbf{e}; \Gamma \vdash \textbf{exit} \supseteq \emptyset}\ [\mathsf{ex}]$$

$$\frac{}{\ell_\mathbf{e}, \ell_\bullet, \ell_\mathbf{s}, \ell_\mathbf{e}; \Gamma \vdash \textbf{end} \supseteq \emptyset}\ [\mathsf{end}] \qquad \frac{}{\ell_\circ, \ell_\mathbf{s}, \ell_\mathbf{s}, \ell_\mathbf{e}; \Gamma \vdash \textbf{start} \supseteq \emptyset}\ [\mathsf{start}]$$

$$\frac{\Gamma(n) = a^\ell}{\ell, \ell, \ell_\mathbf{s}, \ell_\mathbf{e}; \Gamma \vdash \textbf{node}\ n \supseteq \emptyset}\ [\mathsf{lab}] \qquad \frac{\begin{array}{c}\Gamma(n) = a^\ell \\ \ell_\circ, \ell_\bullet, \ell_\mathbf{s}, \ell_\mathbf{e} \vdash [\![p = c_1, \ldots, c_m]\!]^{a^\ell} \supseteq g\end{array}}{\ell_\circ, \ell_\bullet, \ell_\mathbf{s}, \ell_\mathbf{e}; \Gamma \vdash n \supseteq g}\ [\mathsf{cfg}]$$

Figure 2.5 Semantic constraints of the control-flow rules in FlowSpec

two AST nodes, resulting in actual edges in the constraints. Lastly the [cfg] rule handles the recursive call of `cfg`, where it will use any cfg rule from the program which matches the AST node that the variable refers to.

2.4.2 Transfer Functions

Transfer functions for properties come from the property rules in FlowSpec. These rules define $\text{Analysis}_\bullet(\ell)$ in terms of $\text{Analysis}_\bullet(\ell')$. However, there can be multiple matching edges, multiple ℓ'. Therefore, we use $\text{Analysis}_\circ(\ell) = \bigsqcup_{(\ell,\ell') \in F} \text{Analysis}_\bullet(\ell')$ for recursive calls instead. This means that we can map our property rules onto mathematical transfer functions, which is what we

Transfer function mapping $\boxed{\Gamma \vdash R \Rightarrow \mathcal{F}}$

$$\frac{\Gamma \vdash [\![p]\!]^{a^\ell} = \Gamma' \quad \Gamma' \vdash [\![e[n\ n_{adj} := l]]\!] \Rightarrow e_\lambda}{\Gamma \vdash [\![n\ n_{adj} \to p = e]\!]^{a^\ell} \Rightarrow f^n_\ell(l) = e_\lambda}\ \text{[transfw]}$$

$$\frac{\Gamma \vdash [\![p]\!]^{a^\ell} = \Gamma' \quad \Gamma' \vdash [\![e[n\ n_{adj} := l]]\!] \Rightarrow e_\lambda}{\Gamma \vdash [\![n\ p \to n_{adj} = e]\!]^{a^\ell} \Rightarrow f^n_\ell(l) = e_\lambda}\ \text{[transbw]}$$

$$\frac{\Gamma \vdash [\![p]\!]^{a^\ell} = \Gamma' \quad \Gamma' \vdash [\![e]\!] \Rightarrow e_\lambda}{\Gamma \vdash [\![n\ p = e]\!]^{a^\ell} \Rightarrow f^n_\ell(l) = e_\lambda}\ \text{[trans]}$$

Figure 2.6 Mapping of transfer functions in FlowSpec to Monotone Frameworks

Expression semantics $\boxed{\Gamma \vdash e \Rightarrow \mathcal{V}}$

$$\frac{\Gamma \vdash e \Rightarrow v_i}{\Gamma \vdash \textbf{occurrence}(e) \Rightarrow i}\ \text{[occ]} \qquad \frac{\Gamma \vdash \textbf{occurrence}(e) \Rightarrow i}{\Gamma \vdash \textbf{type}(e) \Rightarrow \mathcal{T}_i}\ \text{[type]}$$

$$\frac{\begin{array}{c}\Gamma \vdash \textbf{occurrence}(n_2) \Rightarrow i \\ \vdash n_1\,\{n_2\}_i \in \mathcal{R} \\ \mathcal{S} \vdash n_1\,\{n_2\}_i \longmapsto n_1\,\{n_2\}_j\end{array}}{\Gamma \vdash n_1\texttt{\{}\ n_2\ \texttt{\}} \Rightarrow n_1\,\{n_2\}_j}\ \text{[ref]} \qquad \frac{\begin{array}{c}\Gamma \vdash \textbf{occurrence}(n_2) \Rightarrow i \\ \vdash n_1\,\{n_2\}_i \in \mathcal{D}\end{array}}{\Gamma \vdash n_1\texttt{\{}\ n_2\ \texttt{\}} \Rightarrow n_1\,\{n_2\}_i}\ \text{[decl]}$$

$$\frac{}{\Gamma \vdash n_1(n_2) \Rightarrow n_{1,\bullet}(n_2)}\ \text{[prop]}$$

Figure 2.7 Big-step semantics of a subset of expressions in FlowSpec. An occurrence can only be found on expressions that evaluate to terms from the program, which have an occurrence number i. The static components used are: the set of references $\mathcal{R}$ and declarations $\mathcal{D}$, the scope graph $\mathcal{S}$, and the type relation $\mathcal{T}$.

do in Figure 2.6. Again, we abstract over pattern matching, and we translate expressions into lambda terms to fit the mathematical framework.

We use $\mathcal{F}$ for the transfer function space. The property rules are translated by pattern matching on the AST, then substituting all recursive calls with l, the argument name of the transfer function, and finally translating the functional code into a single mathematical expression.

A mapping from expressions to lambda terms would be a tedious exercise, therefore we separately define the big-step semantics of the interesting part of the expressions in Figure 2.7. In particular we have describe to lookup of occurrences, types and names. The occurrence lookup extracts the occurrence index from a term that originated from the program. Type lookup uses the occurrence index to uniquely identify a term in the program and looks up the

$$
\begin{array}{rlr}
S & ::= \ldots & \text{Sections} \\
 & \mid\ \texttt{lattices}\ \overline{T_L} & \text{Lattice definition section} \\
 & \mid\ \texttt{types}\ \overline{T_T} & \text{Type definition section} \\
 & \mid\ \texttt{functions}\ \overline{T_F} & \text{Function definition section} \\
T_L & ::= n\ \bar{t}\ \texttt{where}\ \overline{L} & \text{Lattice definitions} \\
T_T & ::= n\ \texttt{=}\ n\ \bar{t}\overline{\{\texttt{|}\ n\ \bar{t}\}} & \text{(ADT) Type definitions} \\
T_F & ::= n\ \overline{\{(n\texttt{:}\ t)\}}\ \texttt{=}\ e & \text{Function definitions} \\
L & ::= \texttt{type}\ \texttt{=}\ t\ \mid\ \texttt{lub(}n\texttt{,}\ n\texttt{)}\ \texttt{=}\ e\ \mid\ \texttt{leq(}n\texttt{,}\ n\texttt{)}\ \texttt{=}\ e & \text{Lattice components} \\
 & \mid\ \texttt{bottom}\ \texttt{=}\ e\ \mid\ \texttt{top}\ \texttt{=}\ e\ \mid\ \texttt{glb(}n\texttt{,}\ n\texttt{)}\ \texttt{=}\ e &
\end{array}
$$

Figure 2.8 The types and function part of FlowSpec's grammar.

```
type ConstProp =
  | Top()
  | Const(int)
  | Bottom()

lattice Const where
  type = ConstProp

  lub(l, r) = match (l,r) with
    | (Top(), _) => Top()
    | (_, Top()) => Top()
    | (Const(i), Const(j)) => if i == j then Const(i) else Top()
    | (_, Bottom()) => l
    | (Bottom(), _) => r

  bottom = Bottom()
```

Listing 2.15 A constant propagation type and lattice in FlowSpec. The $\sqsubseteq$ operation is derived from the $\sqcup$ operation by default, although we allow both to be defined.

type in globally available type relation $\mathcal{T}$. Next to type information, we also have access to name information from static analysis, such as scope graph $\mathcal{S}$, set of reference $\mathcal{R}$, and set of declarations $\mathcal{D}$. This provides the information necessary for the two name lookup rules, which together normalise names to their declaration. A declaration is directly found in the $\mathcal{D}$, while a reference in $\mathcal{R}$ is resolved using the scope graph. These normalised names give an intuitive equality semantics for names in FlowSpec: names that resolve to the same declaration are the same.

In the rules for transfer functions we saw that a property rule can use the property information from the neighbouring node. A property can also make use of other properties that have already been calculated. Note that this means that properties cannot depend on each other cyclically. As long as no cyclic dependency exists, we can define a property lookup that uses the property information after the effect of the node ([prop]).

2.4.3 *Lattices*

Users of FlowSpec can define their own algebraic data types and lattice definitions on these types. Of the 5-tuple $(\top, \bot, \sqsubseteq, \sqcup, \sqcap)$, $\sqcap$ and $\top$ are not actually used by the implementation and may be left out of the lattice definition. The other three elements are called `bottom`, `leq` and `lub`. The grammar for this part of the language can be found in Figure 2.8. The lattice definition contains an associated type to that it can be used in any place where a type can be used. We provide an example of a constant propagation lattice in Listing 2.15. Lattices are required in the type position of a data-flow property definition. External property definitions, which may give access to other analysis information such as name sets and type are not required to hold lattices.

2.4.4 Built-in Types and Functions

FlowSpec has the built-in types `Set`, `Map` and `List`, and a number of built-in functions on these types. The `MaySet` and `MustSet` can technically be defined within FlowSpec. However, the `MustSet` needs a symbolic bottom value to represents the largest possible set. For ease of use we make `MustSet` built-in so values from the lattice can be considered sets instead of a union type of sets and the symbolic bottom value.

2.5 Implementation

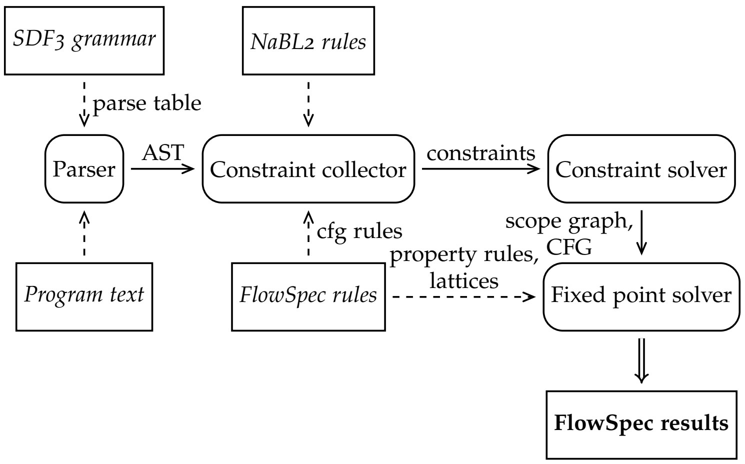
We integrated our implementation of FlowSpec in the Spoofax (Kats and E. Visser 2010) language workbench. Spoofax provides domain-specific meta-languages to declaratively specify a programming language. In this section we provide an overview of how FlowSpec is integrated into Spoofax and what the different parts of the FlowSpec implementation are.

2.5.1 Architecture

Consider Figure 2.9. SDF3 is used for the specification of the grammar and abstract syntax, from which a parse table and different editor services are extracted (Kats, Kalleberg and E. Visser 2010). The parse table is used by the parser in Spoofax to parse program text into an abstract syntax tree (AST). NaBL2 (van Antwerpen et al. 2016) is used from specifying name and type rules, based on the scope graphs (Néron et al. 2015) model that can handle many different binding patterns. With an NaBL2 specification, Spoofax can extract constraints from a program AST, which are fed to a custom constraint solving engine that builds the scope graph.

FlowSpec is active in this same stage. Based on a FlowSpec specification, in particular the control-flow rules, we can automatically extract more constraints from the program AST to build the control-flow graph (CFG). The same constraint solving engine is used, which we adapted to be able to build a control-flow graph. At this point each CFG node is also associated with a transfer function.

The transfer functions, derived from the property rules, are passed to a separate fixed-point solver that we built for FlowSpec, along with the CFG and scope graph. Remember that name information from the scope graph is also used by FlowSpec. The end result is the computed data-flow properties, which



☙ **Figure 2.9** Architecture diagram of FlowSpec within the context of the Spoofax language workbench. A program from an object language is first parsed using a grammar in SDF3. Then we use an NaBL2 static semantics definition to analyse the names and types in the program. The same machinery is used to build the control flow graph, based on the FlowSpec control flow graph rules. A separate fixed point solver is the new addition that computes data-flow information based on the FlowSpec specification.

can be queried in a later stage. These properties are connected to CFG nodes, which are in turn AST nodes, therefore you need only the AST node and the name of the property to request the information.

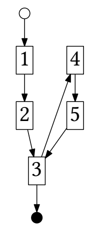
2.5.2 *Control-flow Graph Construction*

The control-flow graph is built in two steps. First the CFG rules from a FlowSpec specification are used to extract edges from the program AST. The edge list is used to create the control-flow graph. At this point the control-flow graph still uses explicit artificial nodes for every `entry` and `exit`.

2.5.3 *Data-flow Solver*

The data-flow solver takes in the CFG, the scope graph and the transfer functions. We apply the transfer functions through an interpreter written in the Truffle (Wimmer and Würthinger 2012) framework.

The simplest version of a solving algorithm for monotone frameworks is a worklist algorithm. All nodes of the CFG are added to the worklist algorithm. When the algorithm computes a new value for a node from the worklist, all out-neighbours of that node in the CFG get re-added to the worklist. Although this is a correct algorithm, it may compute information in an inefficient order when the graph has loops. See Figure 2.10 for a visual example of efficient and inefficient order.



Inefficient	3	2	1	5	4	3	2	1	5	4	3	2	1
Efficient	3			5	4	3			5	4	3	2	1

Figure 2.10 An illustration of efficient and inefficient order in a backward analysis that requires two round through loop 3,4,5 before reaching a fixed point. The inefficient order always propagates from 3 to 2,1 first, whereas the efficient order first propagates from 3 to 5,4.

Strongly Connected Components

We first compute a topological ordering of strongly connected components (SCCs) in the control-flow graph (Horwitz, Demers and Teitelbaum 1987; Jourdan and Parigot 1990). Within each SCC we use a reverse post-order of the depth first spanning forest (Kam and Ullman 1976). This ordering is more efficient in that we can compute fixed points per SCC and only propagate information to other SCCs in the graph afterwards.

It is also designed so the initial $\bot$ value of the lattice is not given to the user-defined transfer functions, it only occurs in lattice operations $\not\sqsubseteq$ and $\sqcup$. This can be important as in general a *must* analysis has the set of all possible values as the bottom of the lattice. This can of course be restricted to a set with all possible values from the program, but such a set would then have to be provided by the analysis author. Instead we can make sure this is not a concern by not exposing $\bot$ to user-defined transfer functions, which allows us to describe $\bot$ symbolically for *must* analysis. The lattice operations have clearly defined laws around $\bot$ without needing to look into the set, so we can implement the operation's cases with $\bot$ symbolically.

The computation of the ordering uses a slightly adapted version of Tarjan's strongly connected components (SCCs) algorithm (Tarjan 1972). The detailed explanation of the adaptation is in Appendix A.1.

Solving Algorithm

As our data-flow properties may (non-cyclically) depend on each other, we order the properties topologically and then solve each one in turn. The algorithm is given in pseudo-code in Listing 2.16.

The first inner loop initialises the property analysis. The extremal labels `starts` of the control-flow graph F' are initialised with the extremal value `Prop.initial`, everything else with $\bot$.

The main loop traverses the topologically ordered strongly connected components (SCCs), and uses a `while` loop to recompute the SCC if the previous iteration changed something. No worklist is necessary as any node can influence any other node.

```
for Prop in topologically ordered Properties:
  for $\ell$ in F:
    $\text{Prop}_\circ(\ell)$ := $\bot$

  if Prop.direction = forward:
    $F'$ := F
  else:
    $F'$ := F.flipped

  for $\ell$ in $F'$.starts:
    $\text{Prop}_\circ(\ell)$ := Prop.initial

  for scc in $F'$.sccs:
    done := false
    while not done:
      done := true
      for $\ell$ in scc:
        for $(\ell, \ell')$ in $F'$.edges:
          if $f_\ell^{\text{Prop}}(\text{Prop}_\circ(\ell)) \not\sqsubseteq \text{Prop}_\circ(\ell')$:
            $\text{Prop}_\circ(\ell')$ :=
              $f_\ell^{\text{Prop}}(\text{Prop}_\circ(\ell)) \sqcup \text{Prop}_\circ(\ell')$
            if $\ell'$ in scc:
              done := false

  for $\ell$ in $F'$:
    $\text{Prop}_\bullet(\ell)$ := $f_\ell^{\text{Prop}}(\text{Prop}_\circ(\ell))$
```

Listing 2.16 Worklist algorithm used in the implementation of FlowSpec. Properties is the list of dataflow properties. F is the control flow graph. f_ℓ^{Prop} is the transfer function of property Prop for control flow graph node ℓ. Lattice operations and values are those corresponding to the lattice of Prop.

The SCC itself is traversed in its reverse post-order. For each node ℓ in the SCC we traverse the outgoing edges (ℓ, ℓ') and use the transferred version of the property at ℓ to see if it would contribute to ℓ'. If so, the transferred property of ℓ is added to the property for ℓ' with the least-upper-bound operator.

After the main loop, the final loop uses the transfer function one more time to calculate the property just after the effect of each control-flow graph node.

Filtering the Control-flow Graph

For every data-flow property, the control-flow nodes have an associated transfer function. Most nodes in the graphs have the identity transfer function, especially `entry` and `exit` nodes. Before the solving algorithm runs, we traverse the graph once to reduce it to only the nodes that actually contribute to the solution. This is especially cheaper when we can remove nodes from a cycle in the graph. Values computed for the previous control-flow graph are propagated to those nodes which have an identity transfer function at the end of the solving phase.

2.6 Case Study of GreenMarl

We evaluate the expressiveness and conciseness of FlowSpec with a number of case studies. So far we have presented our example analyses on a simple imperative language While (F. Nielson, H. R. Nielson and Hankin 2005, pp. 3–4). In our first case studies we expand this toy imperative language to the full language of GreenMarl (Hong, Chafi et al. 2012), a domain specific language for graph processing. First we introduce the domain concepts and the GreenMarl language.

2.6.1 *The Domain of Graph Analysis*

In principle any relational data can be considered a graph, although binary relations are easiest to map onto nodes and edges of a graph. For higher arity relations one may employ a property graph representation, where nodes and edges can be labeled with extra information. The benefit of considering data as a graph is that standard graph algorithms can be applied to extract useful information from the data.

Large datasets from the big data world can be seen and processed as property graphs. But this requires high-performance processing, to handle the large amount of data within a reasonable amount of time. Here the issues that crop up is that a straight-forward implementation of a classic graph algorithm in a general purpose programming language usually is not able to fully exploit modern hardware for computation on large data. Both multi-core processor parallelism and multi-machine parallelism that is usually used for larger data processing requires that algorithms are mixed with bookkeeping and interoperation code, or the algorithm has to be manipulated to fit a framework.

2.6.2 *An Introduction to GreenMarl*

GreenMarl is a domain-specific language for efficient graph analysis (Hong, Chafi et al. 2012). To support its efficiency goal it provides domain-specific and non-domain specific language features so the user can expose opportunities of data-parallelism to the compiler. The style of the language is imperative so graph analysis algorithms can be written in their natural form using graph specific features and imperative loops. The compiler then applies static analysis and outputs highly optimised code for the specified graph analysis.

GreenMarl operates on property graphs, by accepting graphs, node-properties and edge-properties as inputs to its programs, as well as primitive data such as integers, strings and floating point numbers and collections such lists, sets and maps. While the input graph cannot be mutated in GreenMarl, properties can, and new properties can be created on the graph.

The graph can be iterated over using domain-specific ranges, such as the nodes or edges of the graph, or the neighbours of a node. It can also be queried for neighbourhood information. Besides a standard for loop over such ranges, the language provides the parallel foreach loop, and depth- and breadth-first search traversals over graph ranges.

```
1   procedure ccOne(g: graph;
2       cc: nodeProperty<double>) : bool {
3     if(g.numNodes() == 0) { // corner case empty
4       return true;
5     }
6
7     // Kosaraju (simplified)
8     nodeProperty<bool> checked;
9     g.checked = false;
10    node t = g.pickRandom();
11    inDFS(n: g.nodes from t) {
12      n.checked = true;
13    }
14    if(any(v: g.nodes) {!v.checked}) {
15      return false; // not strongly connected
16    }
17    g.checked = false;
18    inDFS(n: g^.nodes from t) {
19      n.checked = true;
20    }
21    if(any(v: g.nodes) {!v.checked}) {
22      return false; // not strongly connected
23    }
24
25    // Closeness Centrality
26    foreach(n: g.nodes) {
27      long levelSum = 0;
28      inBFS(v: g.nodes from n) {
29        levelSum += currentBFSLevel();
30      }
31      n.cc = 1.0 / (double) levelSum;
32    }
33    return true;
34  }
```

Listing 2.17 Closeness Centrality (Unit Length) – Simplified

2.6.3 An Example GreenMarl Program

Consider the GreenMarl program in Listing 2.17. This program computes the Closeness Centrality (Bavelas 1950) measure on a graph, assuming all edges are the same length. Closeness Centrality of a node in a graph is the reciprocal (line 31) of the sum of the shortest paths to every other node in the graph. For unit length edges this can be found by using breadth-first search (lines 28-30) to visit all nodes, using the 'level' of the breadth-first search as the shortest path length. Before the centrality measure is computed, a simplified version of Kosaraju's algorithm for strongly connected components (Kosaraju 1978) is used to check that the input graph is strongly connected. This check initialises a boolean flag for each node. Then using that flag it checks that every node can be reached from a randomly picked node using a depth-first search. The flag is reset and used again, but now the depth-first search is done on the reverse graph.

Within parallel regions, such as the foreach loop and the breadth-first search, the operations are statically checked not to contain data races. The add-and-assign operator is called a reduce-assignment and is explicitly safe to perform in parallel. The language does require that no other writes or reductions with different operators are done within the same parallel section (in this case the breadth-first search).

Next to reduce-assignments, there are reduction expressions. In the example program these are used in the strongly connected check, the `any` expression used in the `if` conditions is a parallel combinator of boolean values.

2.6.4 *The Current GreenMarl Compiler*

The current implementation of GreenMarl already uses the Spoofax language workbench. The GreenMarl compiler uses SDF3 for its grammar definition, and the older NaBL name binding and TS type system languages for its static semantics implementation. The compiler uses the Stratego transformation language to analyse, optimise and generate code.

The current implementation of optimisations often have the enabling analysis embedded in that code. This makes it hard to find out what analysis is necessary, whether some analysis can be reused by other optimisations, and whether the optimisation and analysis are correct. In our search for analyses and optimisations that benefit most from FlowSpec, we have found dead code elimination, constant propagation and loop unswitching[1]. Constant propagation is not implemented in the GreenMarl compiler yet because of time constraints on the compiler development team.

2.6.5 *Control-Flow Graph*

The whole of GreenMarl requires 77 control-flow rules. The rules span 165 lines of code, including comments and empty lines. Listing 2.18 shows a sample of the control-flow graph code. The full list of rules can be found in Appendix A.2.1.

By comparison, back in Listing 2.5 on page 29 we saw 10 control-flow rules for the While language, which took 18 lines of code (again including empty lines). This makes sense, given that on average most language structures have very simple control-flow expressed on a single line, followed by a blank line for readability.

In GreenMarl expressions are not desugared to binary and unary operations that share the same abstract syntax. We chose not to do this desugaring ourselves, which would lead to changes throughout the rest of the compiler. However, if such a change were made, 15 control-flow rules would be reduced to two, and expression related analyses would also shrink in size.

[1]Loop unswitching is an optimisation that pulls a conditional program fragment out of a loop when the condition is loop-independent. After the optimisation the conditional wraps two modified versions of the loop, one to be executed if the condition is true, the other if the condition is false.

```
control-flow rules

  Block(statements) = entry -> statements -> exit
  DeferAssign(lhs, rhs, _) = entry -> rhs -> lhs -> exit
  InReverse(filter, statement) = entry -> filter -> statement -> exit
  InPost(filter, statement) = entry -> filter -> statement -> exit
```

Listing 2.18 A sample of the control flow graph rules for Green-Marl

```
property rules

  live(VarAssign(n) -> next) = live(next) \ { Var{n} }
  live(PropAssign(_,p) -> next) = live(next) \ { Prop{p} }
  live(IterBounds(n, _, _) -> next) = live(next) \ { Var{n} }
  live(XFSIterBounds(n, _, _) -> next) = live(next) \ { Var{n} }
  live(PropRef(_,p) -> next) = live(next) \/ { Prop{p} }
  live(VarRef(n) -> next) = live(next) \/ { Var{n} }
  live(_ -> next) = live(next)
```

Listing 2.19 Live variables analysis for Green-Marl

2.6.6 *Live Variables*

Live variables analysis can be used to perform dead code elimination in the GreenMarl compiler in a more principled way. Currently dead code is discovered in an ad-hoc manner where only variables that are completely unused are removed. These variables are discovered by performing multiple tree traversals over the abstract syntax of a procedure, one to collect all local variables, and then one per variable to check that the variable is not referenced.

The FlowSpec implementation of live variables analysis for GreenMarl in Listing 2.19 is the full analysis. The analysis tracks variables and property variables. A `VarAssign` is a variable reference on the left-hand side of an assignment.

Note that the control-flow rules for GreenMarl grew to 77 rules, compared to the 10 of While, but the rules for live variables analysis only grew from 3 rules to 7 rules.

2.6.7 *Constant Propagation*

The definition of constant propagation follows the approach from Section 2.3.10. The full analysis implementation is available in Appendix A.2.6. Our implementation is 21 rules, each of which takes up 4 lines of code and 1 blank line for readability. Although the implementation is adequate, we find the repetition of similar `match` clauses less concise than ideal. This is an area where we believe we can still improve on the design of FlowSpec.

2.6.8 *Reaching Definitions*

Reaching definitions analysis can be used for many applications where some form of data dependence is required. In this case we define this analysis for GreenMarl for the use case of loop unswitching (Allen and Cocke 1972). This optimisation pulls an if statement out of a loop when the condition of the

```
properties
  reaching: MaySet(term * Option(index))

property rules
  reaching(prev -> this@Decl(n, _, InArg())) =
    { (n, Some(indexOf(this))) } \/ reaching(prev)

  reaching(prev -> this@Decl(n, _, OutArg())) =
    { (n, Some(indexOf(this))) } \/ reaching(prev)

  reaching(prev -> this@Decl(n, _, Local())) = { (n, None()) } \/ reaching(prev)

  reaching(prev -> this@VarAssign(n)) =
    { (n, Some(indexOf(this))) } \/ { (m, l) | (m, l) <- reaching(prev), m != n }

  reaching(prev -> this@PropAssign(_,p)) =
    { (p, Some(indexOf(this))) } \/ { (m, l) | (m, l) <- reaching(prev), m != p }

  reaching(prev -> this@IterBounds(n, _, _)) =
    { (n, Some(indexOf(this))) } \/ { (m, l) | (m, l) <- reaching(prev), m != n }

  reaching(prev -> this@XFSIterBounds(n, _, _)) =
    { (n, Some(indexOf(this))) } \/ { (m, l) | (m, l) <- reaching(prev), m != n }

  // note that we model output effects like printing as writing to an artificial
  variable, which allows reasoning about output dependences
  reaching(prev -> this@Print(_,_)) =
    { (Print(), Some(indexOf(this))) }
      \/ { (m, l) | (m, l) <- reaching(prev), m != Print() }

  reaching(prev -> _) = reaching(prev)
```

Listing 2.20 Reaching definitions analysis for Green-Marl

if statement does not depend on the loop, something that can be discovered with reaching definitions analysis. By interchanging the if statement and loop, the loop does need to be duplicated. One version of the loop for when the if condition is true, and one for when the if condition is false.

This saves the overhead of conditional branching inside the loop, and enables the recognition of other optimisation patterns for the loop. For GreenMarl a particular pattern, that is important when the program is compiled to a distributed setting, is the transfer of data from every node to every neighbour node. This pattern is a loop over all nodes in the graph, and within it only a loop over all neighbours of the node. Therefore if the inner loop is nested in an if statement, loop unswitching can help.

In Listing 2.20 we show the rules of an enhanced Reaching Definitions analysis that explicitly tracks uninitialised variables too. This enhanced analysis would be three rules in an extended version of While with variable declaration. For GreenMarl we have 9 rules, one of which tracks the writing to an output channel instead of a variable to keep track of data dependencies induced by the effect of printing messages.

2.6.9 *Definite Assignment*

We can use definite assignment analysis in GreenMarl for the code generation task of initialisation. When a variable is defined, it is not necessarily initialised. This is particularly of interest for variables that have a collection type, such as a set. Within GreenMarl the semantics is that a defined variable of type set holds an empty set. However, if the variable is later definitely assigned a set, the variable does not need to be initialised.

We can use the results of reaching definitions analysis for this definite assignment analysis. The reaching definitions analysis we defined previously tracks variables from definition, marking these as uninitialised. Wherever a `(n, None())` pair is in the set, n is not definitely assigned there.

2.6.10 *Available and Very Busy Expressions*

Available expressions and very busy expressions are very similar in definition, as we observed previously. We present available expressions analysis for GreenMarl in Listing 2.21. The definition of the analysis is not particularly short, since all expressions have distinct abstract syntax that needs to be handled in a separate rule. Compared to the definition for While, which needed only 3 rules, this is a rather steep increase to 24 rules. However, as we have noted before, this is due to the shape of the AST that the GreenMarl compiler works with, which was outside of our control.

2.6.11 *Performance Measurement*

Although we do not have analyses to compare against, we can measure the current performance of the analyses we presented on typical GreenMarl. FlowSpec was designed to be a concise, *executable* specification language, where we would like the execution to be of practical use. Therefore we are not after best-in-class performance, but FlowSpec should have a reasonable performance for typical programs.

Setup

Our test machine has a 2.8 GHz Intel Core i7 processor, with 16 GB 1600 MHz DDR3 RAM. It runs MacOS 10.14.2. The Java version is 1.8.0_152-b16, run on the HotSpot VM 25.152-b16. We use JMH, the OpenJDK benchmark harness library, version 1.21. Each benchmark is run with JVM arguments `-Xms512m -Xmx2g -Xss16m`, meaning the initial JVM heap size is 512 MiB, the maximum JVM heap size is 2 GiB and the JVM thread stack size is 16 MiB.

We run 5 warmup iterations, 10 seconds each, after which we run 5 measurement iterations. The benchmark sets up all required dependencies beforehand.

Inputs

We gathered three typical size GreenMarl programs: Closeness Centrality[2], Closeness Centrality with edge weights, and Betweenness Centrality[3]. The characteristics of these inputs are in Table 2.1, where lines of code (LOC) are

[2]as previously shown in Listing 2.17 on page 46

[3]Gathered from (Oracle Corporation 2015b)

```
properties
  available: MustSet(term)
  external refs: Set(name)

property rules
  available(prev -> _) = available(prev)

  available(prev -> VarAssign(n)) =
    { expr |
      expr <- available(prev),
      !(Var{n} in refs(expr)) }

  available(prev -> PropAssign(_,p)) =
    { expr |
      expr <- available(prev),
      !(Prop{p} in refs(expr)) }

  available(prev -> this@IterBounds(n, _, _)) =
    { expr |
      expr <- available(prev),
      !(Var{n} in refs(expr)) }

  available(prev -> this@XFSIterBounds(n, _, _)) =
    { expr |
      expr <- available(prev),
      !(Var{n} in refs(expr)) }

  available(prev -> this@Abs(_)) = available(prev) \/ { this }
  // Eliding similar lines for UMin, Mul, Div, Mod, Add, Sub, Not,
  //  Or, Eq, Gt, Lt, Geq, Leq, Neq, Cast, TerIf, FuncCall, ProcCall
```

Listing 2.21 Available expressions analysis for Green-Marl

Input name	LOC	Bytes of input
CC	22	1892
BC	17	2689
CC weighted	41	6963

Table 2.1 Benchmark inputs and their characteristics.

measured without blank lines and comments, and the bytes of input are in the intermediate representation on disk from which they are read for input to the benchmarks.

Results

We present the measurement results in Table 2.2, where each number is the arithmetic mean of the five measurements for that benchmark. Given the low analysis time for these typical size inputs, we are confident that FlowSpec analyses can be effectively used within the GreenMarl compiler. In Section 2.7.4 we mention some more optimisation options we have to improve the performance of FlowSpec even further.

Input name	Live Variables	Reaching Definitions
CC	0.9	1.4
BC	1.3	2.8
CC weighted	2.7	10.7

Table 2.2 Analysis time in milliseconds for implementations of Live Variable and Reaching Definitions in FlowSpec on typical GreenMarl programs.

2.7 Case Study of Stratego

We evaluate the expressiveness and conciseness of FlowSpec with a number of case studies. So far we have only presented example analyses for typical imperative programming languages. In this section we present a case study of an analysis written in FlowSpec for a programming language with a different paradigm: the Stratego (Bravenboer, Kalleberg et al. 2008) term rewriting language. We show how we specified a reaching definitions analysis in FlowSpec and compare it to the existing implementation of reaching definitions in the Stratego compiler, which is itself written in Stratego. We compare not only the implementation from a source code perspective but also give a performance comparison between the FlowSpec implementation and the implementation in Stratego.

First we introduce the domain concepts and the Stratego language.

2.7.1 *Term Rewriting and Stratego*

Stratego is a language for program transformation. The language has features for defining *rewrite rules*, and *strategies* for the application of those rewrite rules. Given an Abstract Syntax Tree that presents a program, a Stratego program can transform terms from the tree with rewrite rules and apply them in the right places in the right order with strategies.

Any rule or strategy can fail to apply. Special strategy combinators allow recovery from failure, which looks similar to an if-else but based on failure or success instead of a boolean value. Rewrite rules can be expressed as strategies, and in fact the Stratego compiler desugars all features down to a core language that consists only of strategies.

Stratego Core. Stratego core consists of a list of strategy definitions. Each strategy definition has zero or more strategy arguments (functions, making the strategy higher-order), zero or more term arguments (data), and an implicit argument: the current term. The body of a strategy definition is a strategy expression, which can consist of:

1. `fail`, the primitive strategy that fails
2. `id`, the primitive strategy that succeeds and does nothing to the current term
3. pattern-match, matches a pattern against the current term, possibly failing or binding variables when it succeeds
4. pattern-build, replaces the current term with another term, possibly failing when using an unbound variable

```
control-flow rules
  root SDefT(_, sargs, targs, body) =
    start -> sargs -> targs
          -> body -> end

  node VarDec(_, _)

  Let(defs, body) =
    entry -> defs -> body -> exit

  SDefT(_, sargs, targs, body) =
    entry -> exit,
    entry -> sargs -> targs
          -> body -> exit

  CallT(_, sargs, targs) =
    entry -> targs -> this -> exit,
    this -> sargs -> exit

  node Fail()
  node Id()
  node Match(_)
  node Build(_)

  Scope(_, body) =
    entry -> this -> body -> exit

  Seq(first, second) =
    entry -> first -> second -> exit

  GuardedLChoice(c, t, e) =
    entry -> c -> t -> exit,
    entry -> e -> exit

  Some(s) =
    entry -> this -> exit,
             this -> s -> exit

  One(s) =
    entry -> this -> exit,
             this -> s -> exit

  All(s) =
    entry -> this -> exit,
             this -> s -> exit
```

Listing 2.22 Most of the control flow graph rules for Stratego

5. scope, defines a scope with a list of fresh (unbound) variables
6. sequence, apply one strategy expression after the other
7. guarded choice, the if-then-else lookalike based on failure instead of boolean values
8. `one`, `some`, `all`, three language constructs that take a strategy argument and apply that strategy on one, some or all of the children of the current term.
9. strategy call, to call another named strategy
10. let, to define local strategies

2.7.2 *Control-Flow*

With the list of constructs, we are able to define the control-flow of Stratego core, as seen in Listing 2.22.

Note how the guarded choice rule specifies two paths, one for the condition and then branch, one for the else branch *without* the condition. This is something particular to Stratego, where variable bindings from the condition are backtracked when the condition fails at some point.

All call-like constructs have control-flow that optionally goes into the strategy arguments. This models the uncertainty of whether those strategy expressions are executed or not.

2.7.3 *Reaching Definitions*

Reaching definitions is used in a number of places in the current Stratego compiler, which is written in Stratego. Whether a variable is guaranteed to be bound or unbound at some point in a Stratego program is valuable to given errors messages (e.g. on build a pattern with an unbound variable) and to

```
properties

  reachingDefinitions:
    MaySet(name * Option(position))

property rules

  reachingDefinitions(_.start) = {}

  reachingDefinitions(prev -> v@VarDec(n, _)) =
    reachingDefinitions(prev)
    \/ {(Var{n}, Some(position(v)))}

  reachingDefinitions(prev -> Scope(names, _)) =
    reachingDefinitions(prev)
    \/ { (Var{n}, None()) |
         n <- Set.fromList(names) }

  reachingDefinitions(prev -> m@Match(t)) =
    { (n, p) |
      (n, p) <- rdprev,
      !(n in pv && p == None()) }
    \/ { (nm, Some(position(m))) |
         nm <- pv,
         (nm, None()) in rdprev }
  where rdprev = reachingDefinitions(prev)
        pv = patternVars(t)
```

Listing 2.23 Reaching definitions analysis for Stratego

generate efficient code (e.g. elide a check and variable binding code when pattern matching against an always bound variable).

Our reaching definitions analysis for Stratego, written in FlowSpec is given in Listing 2.23. We start without bindings, add reaching definitions for arguments to strategies, add uninitialised variables for scopes, and replace uninitialised variables with their initialisation when they are first matched.

For comparison, the original analysis consists of 232 lines of Stratego code (provided in Appendix A.3) that implement reaching definitions analysis under the name `bound-unbound-vars` in the current Stratego compiler. Our implementation in FlowSpec is only 19 lines.

The Stratego implementation uses a feature called dynamic rules (Bravenboer, van Dam et al. 2006) to implement both the data-flow analysis and specify the name and scope rules for Stratego in an ad-hoc fashion. Other analyses in the Stratego compiler repeat this name and scope structure in a similar way. Notably, this code analyses a subset of Stratego but slightly more than Stratego core. This is most likely a legacy code issue. Currently this analysis is called within the compiler at a point where the AST has been reduced to Stratego core already.

2.7.4 *Performance Comparison*

Since we have a Stratego implementation and FlowSpec implementation of the same analysis, we can do a performance comparison.

Input name	LOC	Strategy definitions	Bytes of input
incremental	105	1	11577
libspoofax	1733	278	168539
libstratego	6071	1891	1169465
libstrc	9841	2690	2341797

Table 2.3 Benchmark inputs and their characteristics.

Setup

Our test machine has a 2.8 GHz Intel Core i7 processor, with 16 GB 1600 MHz DDR3 RAM. It runs MacOS 10.14.2. The Java version is 1.8.0_152-b16, run on the HotSpot VM 25.152-b16. We use JMH, the OpenJDK benchmark harness library, version 1.21. Each benchmark is run with JVM arguments `-Xms512m -Xmx2g -Xss16m`, meaning the initial JVM heap size is 512 MiB, the maximum JVM heap size is 2 GiB and the JVM thread stack size is 16 MiB.

We run 5 warmup iterations, 10 seconds each, after which we run 5 measurement iterations. The benchmark sets up all required dependencies beforehand. The actual measured code is the Stratego strategy in the Stratego case and the FlowSpec analysis in the FlowSpec case.

Inputs

We gathered input program of different size, from the typical size for the incremental Stratego compiler (a single strategy), up to the largest Stratego library we know of. The characteristics of these inputs are in Table 2.3, where lines of code (LOC) are measured without blank lines and comments, and the bytes of input are in the intermediate representation on disk from which they are read for input to the benchmarks.

Results

We present the benchmark results in Table 2.4, where each number is the arithmetic mean of the five measurements for that benchmark.

FlowSpec has reasonable execution times for typical workloads. We do see the execution time grow rather quickly of very large workloads. We see that from the libspoofax input to the libstratego input is a 7x growth in raw bytes of input, the Stratego implementation of the analysis spends 8x the time on that input, but the FlowSpec implementation spends 26x milliseconds.

We are aware of some of the causes for this behaviour. FlowSpec builds up and saves all control-flow graphs to save these as part of the analysis results, as well as the data-flow analysis information. In contrast, the Stratego implementation is manually written in such a way that the analysis information is used and forgotten as soon as possible. FlowSpec also currently runs an AST interpreter for the data-flow rules, which takes a majority of time. The Stratego implementation is compiled entirely.

Correctness. We compared the outcomes of the two analyses to each other. The Stratego implementation immediately transforms the AST by annotating

Input name	FlowSpec	Stratego
incremental	6.65	0.27
libspoofax	148.58	4.49
libstratego	1829.44	33.27
libstrc	6322.17	73.39

Table 2.4 Analysis time in milliseconds for our implementation in FlowSpec and the optimised Stratego implementation in the current Stratego compiler.

each variable use with its estimated state: bound, unbound, or maybe bound. Once the FlowSpec analysis is finished, the code to add such annotations based on the FlowSpec analysis is trivial (just 23 lines of Stratego code).

Threats to Validity

Although this is a small benchmark, more for the sake of curiosity than validation, we still address the threats to validity of the measurements.

External Validity. A threat to the generalisability of these measurements is how we compared only a single analysis (Reaching Definitions) with a single language (Stratego). In fact, intra-procedural analysis of Stratego gives rise to acyclic control-flow graphs. This is not at all representative of typical control-flow graphs. However, this was the analysis and language that were easily available for comparison against an older implementation.

Internal Validity. The comparison we make here is that of end-goal usage, not exactly the same analysis. The Stratego analysis is both analysis and transformation, combined by hand. This combination is a specialisation that does well in performance, although we argue that it is not good for maintainability. On the other hand, FlowSpec computes and returns a control-flow graph, and computes and returns all Reaching Definitions information for the entire program. This is more work, more information that can be used for multiple purposes. And yet the result is not the transformed program.

Construct Validity. Finally, we measure performance on the JVM and need to consider JIT compilation and garbage collection. Thankfully the JMH framework takes care of warmup for the JIT and garbage collection between iterations. We could not eliminate background noise entirely, but all measurement iterations looked to be close to each other. The biggest open question is that of the three phases benchmarked separately which do not sum up to the whole benchmark.

2.8 Related Work

We will shortly discuss the history of monotone frameworks which underlies our work, and some other systems and formalisms for implementing data-flow analysis. Some of the aspects we discuss are summarised in Table 2.5.

System	Scope	Flow-Sensitive	Lattices	Boilerplate	Incremental
FlowSpec	*Procedure*	*Yes*	*Arbitrary*	*Very low*	*No*
JastAdd	Program	Yes	Arbitrary*	High	Yes
Silver	Program	Yes	Arbitrary	Very low	No
Aster	Program	Yes	Arbitrary	Low	No
Stratego	Program	Yes	Arbitrary	Medium	No
Kiama	Program	Yes	Arbitrary	Medium	No
Doop	Program	No	May/Must	High	No
Flix	Program	Yes	Arbitrary	Low	No
MPS-DF	Program	Yes	Arbitrary	Low	No
IncA	Program	Yes	Arbitrary	Low	Yes
Rascal	Procedure	Yes	Arbitrary	Medium	No
CAnDL	Procedure	Yes	N/A	Low	No

* Complex lattices may need to be defined in Java

Table 2.5 Related work summary table.

2.8.1 Monotone Frameworks

Monotone data-flow analysis frameworks (Kam and Ullman 1977) were first introduced as a generalisation over Killdall's lattice theoretic approach to data-flow analysis (Kildall 1973). By replacing the distributivity requirement with a monotonicity requirement for the transfer function, Kam and Ullman found a way to describe more flow problems in a framework with a clear solution by maximal fixed point. This maximal fixed point can be iteratively computed with a simple worklist algorithm. As mentioned in Section 2.2, FlowSpec is based on this analysis framework.

2.8.2 Attribute Grammars

JastAdd. The JastAdd system (Ekman and Görel Hedin 2007b) supports attribute grammars (Knuth 1968) extended with a number of special attributes which allows a declarative intra-procedural control- and data-flow analysis specification (Söderberg, Ekman et al. 2013). In particular, these are reference attributes (Görel Hedin 2000) for control-flow graph (CFG) edges, higher-order attributes (Vogt, Swierstra and Kuiper 1989) for virtual CFG nodes, used for entry/exit of methods, circular attributes (Magnusson and Görel Hedin 2007) for fixed points of data-flow equations, and collection attributes (Magnusson, Ekman and Gorel Hedin 2007) e.g. for the CFG where there are multiple successors.

Note that each feature can be used for data-flow analysis but is not specifically designed for it. Therefore JastAdd is a much more general computation system that has much more expressive power than FlowSpec. The downside of this generality, versus the domain-specific nature of FlowSpec, is the verbosity. Whereas in FlowSpec our specifications are small and the language provides domain terms for each of the features, JastAdd requires an encoding in the different attributes. It is also not clear to us whether higher-order attributes are enough to encode arbitrary lattices. If not, JastAdd's Java integration would be required to implement the lattice.

Silver. Silver (Van Wyk et al. 2010) is another attribute grammar system and specification language that supports similar features to the JastAdd system. However, for control- and data-flow analysis, it provides dedicated syntax which translates to a control-flow graph and temporal logic formulae (CTL-FV) that are offloaded to a model checker (NuSMW). Temporal logic can express reasoning in terms of time, which can be used to express data-flow properties in a declarative manner.

Temporal logic is a very terse notation for data-flow specification, and is subjectively not very easy to read. Our language design for FlowSpec is a very different approach which is not rooted in logic formulae. Instead we use domain names for keywords and provide a declarative approach to specification which borrows from functional programming.

The Silver publication did not report on the performance of their data-flow analysis approach. Model checkers have a sweet spot where their heuristics perform well, but ultimately cannot cover the entire NP-Hard problem space. As such, it may suddenly perform poorly for the problems that Silver generates for it based on the translation Silver uses, the temporal logic formula in the Silver specification, or the particular input program.

Aster. The Stratego strategic programming language was extended with attribute grammars in Aster (Kats, Sloane and E. Visser 2009). Aster allows for attribute decorators that allow the user to program different attribute grammar extensions, which allows it to support declarative flow analysis similar to JastAdd.

Stratego. The Stratego programming language was also directly applied to data-flow analysis by leveraging its dynamic rewrite rules (Bravenboer, van Dam et al. 2006). In this paper the authors apply a combination of rewrite rules and dynamic rules for dynamic propagation of information. Dynamic rules can use either union or intersection to follow control-flow that splits and merges. At the splitting point the dynamic rule is copied to both branches. In all other places dynamic rules are mutated, which is not an issue as the rewrite based on the dynamic information is done immediately. Fixed point calculation can also be done with a similar choice of union or intersection.

In FlowSpec we treat data-flow analysis as a separate concern. In contrast an analysis in Stratego is usually interspersed in the transformation code, which makes the code more difficult to read and understand. This code style also brings frustrations when an analysis already interleaved in a transformation turns out to be more generally useful in other transformations. Extracting and reusing such an analysis is not well supported by dynamic rules. FlowSpec is built around the idea of simple, separate specifications of data-flow analysis. The analysis results can be used to inform an arbitrary amount of transformations.

Kiama. Kiama (Sloane, Roberts and Hamey 2014) is a language processing library in Scala, based on attribute grammars and strategic programming. The interesting property Kiama has over Aster is the provisions for updating analyses after transformation, a concern we currently do not address in FlowSpec. The tree transformations done with strategic programming can invalidate the values of certain attributes that are dependent on the parents of a tree node (e.g. inherited attributes), or some other context. To easily combine attribute grammars with strategic programming, Kiama provides tree-indexed attribute families. The root of the particular tree is used for indexing whenever an attribute is context-dependent.

2.8.3 Relational Programming

Doop. The Doop framework (Bravenboer and Smaragdakis 2009) uses a Datalog dialect for a declarative specification of static analyses such as context-sensitive pointer analysis. In a recent tutorial (Smaragdakis and Balatsouras 2015, p. 46), Smaragdakis and Balatsouras explain different techniques specific to pointer analysis with Datalog examples. These mostly focus on whole-program, flow-insensitive may-analyses. Flow-sensitive analyses and must-analyses are significantly more complex and harder to ensure soundness of.

FlowSpec focusses on a complementary set of data-flow analysis. Instead of whole-program (inter-procedural) flow-insensitive may-analyses, FlowSpec provides support for local (intra-procedural) flow-sensitive analyses with arbitrary lattices.

Flix. The Flix programming language (Madsen, Yee and Lhoták 2016) is a new contender that extends Datalog to a language with user-defined lattices, and monotonic transfer and filter functions on these lattices. These allow Flix to express data-flow analysis with infinite value domains while keeping guaranteed termination with a unique minimal model; under the assumption that the user-defined lattices and functions are defined correctly.

User-defined types and lattices in Flix and FlowSpec are very similar. FlowSpec benefits from the larger Spoofax ecosystem, to develop features such as the (experimental) automatic name abstraction. One may be able to provide name and scope information along with an input program in Flix, and use explicit filtering, but to our knowledge there is no way to automatically filter names that go out of scope.

2.8.4 Other Analysis Approaches

MPS-DF. The MPS language workbench[4] has MPS-DF, an extensible framework for definition of data-flow analyses (Szabó, Alperovich et al. 2016). MPS-DF has support for building data-flow graphs (control-flow graphs with *read* and *write* primitives), and a syntax for writing transfer and confluence operators. These operators form the ingredients that allows MPS-DF to apply a classical monotone frameworks solution. The analysis can be done in an intra-procedural fashion by correctly implementing the operators to abstract over the possible effects of a procedure call, or inter-procedurally by inlining

[4]`https://www.jetbrains.com/mps/`

method calls. To support this variability, two different data-flow graph builders need to be implemented for a procedure call element in the AST.

IncA. Another MPS related language is IncA (Szabó, Erdweg and Völter 2016), a DSL for incremental program analysis. This DSL is built upon the InQuery engine which supports incremental computations using first order logic extended with the least fixed point operator. The language originally only worked for analyses that can be modelled with relations (i.e. may- and must-analysis). It did not support the generation of data that is not directly from the program, such as building intervals in an interval analysis. This was remediated by extending the incremental algorithm for lattice based values (Szabó, Völter and Erdweg 2017). The language design of the IncA DSL evokes a procedural style, whereas FlowSpec uses a declarative style with domain terms.

FlowSpec makes a different trade-off than IncA. We do not support incremental analysis, thereby also avoiding the prohibitive memory overhead of IncA, which the authors mention as a concern for future work. The benefit of IncA is that data-flow analysis can be used for rapid feedback to a user in an IDE setting. This fits well with inter-procedural analysis. With FlowSpec our focus has been on analyses that inform optimisation, which are done in a compiler backend.

Rascal. Rascal (Klint, van der Storm and Vinju 2009) provides a facility for control-flow graph construction with DCFlow (Hills 2014), a domain-specific language. It simplifies the definition of simple control-flow constructions, but does not support abrupt termination such as exceptions. To implement these constructs the user needs to fall back on the DCFlow library in Rascal. Similarly, the actual implementation of data-flow algorithms on top of a CFG is still done in the Rascal language, without a special library or framework for the use-case.

FlowSpec support both control-flow graph construction and data-flow analysis within the domain-specific language. Rascal, as a general-purpose language, can support anything, but without extra support for the use case of data-flow analysis.

CAnDL. The domain specific language CAnDL (Ginsbach, Crawford and O'Boyle 2018) provides a constraint based approach to compiler analysis for LLVM. It is specifically designed for the LLVM intermediate representation, which is in single static assignment (SSA) form. The focus is on programmer productivity, and in their evaluation the authors show several real world use cases where analyses were expressed more briefly in CAnDL than in the original C++ of LLVM or in Polly, a polyhedral optimisation framework.

FlowSpec makes no assumptions on the representation of the program or intermediate representation it analyses. We provide a language parametric analysis DSL, where we cannot make assumptions about a representation such as SSA form. Instead of the specific constraints of CAnDL, we provide property rules where the user can propagate information of their choosing.

2.9 Conclusion

We have presented FlowSpec, a declarative specification language for the domain of data-flow analysis. We have shown its static semantics, and its dynamic semantics as a mapping onto monotone frameworks. The implementation of FlowSpec is integrated into the Spoofax language workbench where it can access name information to take into account during analyses. We have demonstrated a number of example specifications in FlowSpec. We have also demonstrated FlowSpec in a case study of an industrial domain-specific language with domain-specific analyses and a case study of a term transformation language.

In short, FlowSpec provides domain-specific, integrated support for data-flow analysis in compilers. With it, we can remove ad-hoc analyses and provide more maintainable compilers in the future.

Chapter 3

Incremental Compilers with Internal Build Systems

Abstract. Compilation time is an important factor in the adaptability of a software project. Fast recompilation enables cheap experimentation with changes to a project, as those changes can be tested quickly. Separate and incremental compilation has been a topic of interest for a long time to facilitate fast recompilation.

Despite the benefits of an incremental compiler, such compilers are usually not the default. This is because incrementalisation requires cross-cutting, complicated, and error-prone techniques such as dependency tracking, caching, cache invalidation, and change detection. Especially in compilers for languages with cross-module definitions and integration, correctly and efficiently implementing an incremental compiler can be a challenge. Retrofitting incrementality into a compiler is even harder. We address this problem by developing a compiler design approach that reuses parts of an existing non-incremental compiler to lower the cost of building an incremental compiler. It also gives an intuition into compiling difficult-to-incrementalise language features through staging.

We use the compiler design approach presented in this chapter to develop an incremental compiler for the Stratego term-rewriting language. This language has a set of features that at first glance look incompatible with incremental compilation. Therefore, we treat Stratego as our critical case to demonstrate the approach on. We show how this approach decomposes the original compiler and has a solution to compile Stratego incrementally. The key idea on which we build our incremental compiler is to *internally* use an incremental build system to wire together the components we extract from the original compiler.

The resulting compiler is already in use as a replacement of the original whole-program compiler. We find that the incremental build system inside the compiler is a crucial component of our approach. This allows a compiler writer to think in multiple steps of compilation, and combine that into an incremental compiler almost effortlessly. Normally, separate compilation à la C is facilitated by an *external* build system, where the programmer is responsible for managing dependencies between files. We reuse an existing sound and optimal incremental build system, and integrate its dependency tracking *into* the compiler.

The incremental compiler for Stratego is available as an artefact along with this article. We evaluate it on a large Stratego project to test its performance. The benchmark replays edits to the Stratego project from version control. These benchmarks are part of the artefact, packaged as a virtual machine image for easy reproducibility.

Although we demonstrate our design approach on the Stratego programming language, we also describe it generally throughout this chapter. Many currently used programming languages have a compiler that is much slower than necessary. Our design provides an approach to change this, by reusing an existing compiler and making it incremental within a reasonable amount of time.

Based on: Jeff Smits, Gabriël Konat and Eelco Visser (2020). 'Constructing Hybrid Incremental Compilers for Cross-Module Extensibility with an Internal Build System'. In: *The Art, Science, and Engineering of Programming* 4.3, 16. DOI: 10.22152/programming-journal.org/2020/4/16

3.1 Introduction

Compilation time of a software project is an important factor in how easily the project can be changed. When the compilation time is low, it is cheap to experiment with changes to the project, which can be tested immediately after recompilation. Therefore, fast recompilation has been a topic of interest for a long time (Backus and Heising 1964, p. 384).

Already in the early times of FORTRAN (as of FORTRAN II (Backus and Heising 1964, p. 384)), *independent compilation* of modules was used to speed up recompilation. This allowed a change in one module to require only the recompilation of that module and linking against the previously compiled other modules of the program. Skipping the compilation of all modules except the changed one was a significant improvement over compiling everything again. This did come at the cost of possible link-time issues. To be able to link the program together, the modules needed to be up-to-date with the data layout defined in `COMMON`. This was done manually and only checked by the programmer.

Mesa introduced *separate compilation*, which solved this issue by moving the static checks of cross-module dependencies to compile-time (Geschke, Jr. and Satterthwaite 1977). When a Mesa module is compiled, the result includes a symbol file that can be used during the compilation of other modules that depend on that module. Other languages such as ML and C use interface files that are written by the programmer. Separate compilation brought back type correctness for statically typed programs, while preserving fast recompilation from *independent compilation*.

To further speed up recompilation, we can save intermediate results during compilation. If parts of the program do not change, then the intermediate results of those parts can be reused. The term *separate compilation* applies to compilation where intermediate results are saved per file (Geschke, Jr. and Satterthwaite 1977). For sub-file level tracking of changes and intermediate results, the term *incremental compilation* is used (Ryan, Crandall and Medwedeff 1966; Reiss 1984).

Incremental compilation has clear benefits for recompilation speed. However, to make a compiler incremental, we must split the compilation process into separate (ideally independent) parts, track dependencies between these parts, cache previous results, perform cache invalidation, persist caches to disk, detect changes, and propagate them. Unfortunately, this is far from trivial, as these techniques are complicated and error-prone, and cross-cut the concerns of the compiler. Especially in a compiler for a language requiring cross-module linking and integration, correctly and efficiently implementing an incremental compiler can be challenging.

In this chapter, we present an approach to the design of incremental compilers that separates the language-specific aspects of a compilation schema from the language-independent aspects of tracking the impact of a change on the output of compilation. Our compilation method is a hybrid of separate and incremental compilation. It is separate compilation in the sense that we process a changed file in its entirety. However, instead of producing a single intermediate representation for a file, we split the intermediate representation into smaller units (e.g. top-level definitions) and separate summaries for static analysis. By splitting up the file early, we can then proceed to process each unit separately. Changing one of the units within a file will only require the reading and splitting up of that file, after which further processing can proceed incrementally. We use PIE (Konat, Steindorfer et al. 2018; Konat, Erdweg and E. Visser 2018) – an efficient, precise, and expressive incremental build system – to tie together the components of the compiler in order to efficiently propagate changes through the compiler pipeline. We have developed this approach to incrementalise the compiler of Stratego (Bravenboer, Kalleberg et al. 2008; Bravenboer, van Dam et al. 2006), a term rewriting language with open extensibility features. Our approach allows us to reuse almost all of the existing compiler while gaining great improvements in recompilation speed. We evaluate the incremental Stratego compiler with a benchmark on the version control history of a large Stratego project.

In summary, the paper provides the following contributions:

- We present a general design approach for constructing a hybrid separate/incremental compiler from compiler components and a build system.
- We present an application of the approach in the design and implementation of a hybrid separate/incremental compiler for the Stratego transformation language, for which previously only a whole program compiler was available.
- We evaluate the performance of the Stratego incremental compiler applied to a version history of the WebDSL code base. While the from-scratch time of the incremental compiler is longer than the original compiler, the recompilation time of the incremental compiler is less than 10 % of the from-scratch time in most cases.

Outline. We proceed as follows. In the next section we discuss the open extensibility features of the Stratego language, their application to modular language definition, and indicate how these features affect separate/incremental compilation. In Section 3.3, we analyse the interaction between compilers and build systems in general, and discuss the whole program Stratego compiler and a previous attempt at making it incremental. In Section 3.4, we describe our incremental compilation design approach and its application to Stratego. In Section 3.5, we compare the performance of the incremental compiler to the performance of the original, whole-program Stratego compiler, and demonstrate the effectiveness of our solution for successive compilations. In Section 3.6, we discuss related work.

3.2 Open Extensibility in Stratego

The goal of the *expression problem* as formulated by Wadler (Wadler 1998) "is to define a datatype by cases, where one can add new cases to the datatype and new functions over the datatype, without recompiling existing code, and while retaining static type safety (e.g., no casts)." The example used to illustrate the expression problem is the modular definition of a language consisting of a data type for its abstract syntax and operations such as evaluation and pretty-printing on that data type. The problem illustrates the difference in capabilities between object-oriented programming, in which data extension is easy, and functional programming, in which extension with operations is easy.

The Stratego transformation language was designed to support modular language definition with open extensibility. A module can extend a language definition with new AST constructors and/or with new operations, as we illustrate below. Alas, the language design does not count as a solution to the expression problem since it requires whole program compilation and the language is dynamically typed. In this chapter, we address the problem of incrementally compiling Stratego programs in the face of cross-module extensibility of operations (aka strategies), bringing it closer to a solution of the expression problem. In this section, we illustrate the use of open extensibility in Stratego, we discuss the language features that enable that and how they affect separate/incremental compilation, and we discuss a real world application of open extensibility in the WebDSL compiler.

An Example. We illustrate Stratego's extensibility with a fragment from the TFA example language borrowed from Visser (E. Visser 2005). The language definition in Figure 3.1 is organised in a matrix where each cell corresponds to a Stratego module. The columns correspond to language aspects and the rows define data types or transformations. The modules in the top row define the abstract syntax of a language aspect: a core language with variables and function calls, arithmetic expressions, and control-flow constructs[1]. The modules in the second row define the desugar transformation. Arithmetic operator applications are desugared to function applications (taking the AST constructor as function name, using the pattern `f#(ts)` that generically deconstructs a term into its constructor `f` and child terms `ts`), for loops are desugared to while loops with the appropriate initialisation, and if-then statements are desugared to if-then-else statements. The modules in the third row define the eval transformation using rewrite rules for basic reductions and a strategy to define evaluation order. (The details are not relevant for the topic of the paper.) The TFA example described by Visser (E. Visser 2005) provides more (interesting) language extensions and operations, but Figure 3.1 demonstrates the essential extensibility features.

The example illustrates how we can orthogonally extend a language definition with new constructors and/or transformations. We discuss the language features that enable this extensibility.

Modules. Stratego programs are organised in modules, defined in separate files. A module can import other modules, making their contents accessible. Imports are transitive. Modules can define algebraic data type signatures, rewrite rules, and strategies.

Strategic Rewriting. In Stratego, transformations are defined using term pattern matching (`?p`) and term pattern instantiation (`!p`) as basic operations (also

[1]One can argue with the choice of core language, but that is not the topic of this chapter.

$$
\begin{aligned}
p &::= b\\
b &::= \texttt{begin}\ \overline{st}\ \texttt{end}\\
st &::= \texttt{var}\ x\texttt{:}t\texttt{;}\ \mid\ x\texttt{:=}e\texttt{;}\\
&\quad\mid\ f\texttt{(}e_1\texttt{,}\ldots\texttt{,}e_n\texttt{);}\ \mid\ b\\
e &::= x\ \mid\ f\texttt{(}e_1\texttt{,}\ldots\texttt{,}e_n\texttt{)}\\
t &::= \texttt{void}
\end{aligned}
$$

$$
\begin{aligned}
i &::= [0-9]+\\
e &::= i\ \mid\ e_1\texttt{+}e_2\ \mid\ e_1\texttt{*}e_2\\
&\quad\mid\ e_1\texttt{\&}e_2\ \mid\ e_1\texttt{|}e_2\ \mid\ \ldots\\
t &::= \texttt{int}
\end{aligned}
$$

$$
\begin{aligned}
st ::= &\ \texttt{if}\ e\ \texttt{then}\ \overline{st}\ \texttt{else}\ \overline{st}\ \texttt{end}\\
\mid &\ \texttt{if}\ e\ \texttt{then}\ \overline{st}\ \texttt{end}\\
\mid &\ \texttt{while}\ e\ \texttt{do}\ \overline{st}\ \texttt{end}\\
\mid &\ \texttt{for}\ x\ \texttt{:=}\ e_1\ \texttt{to}\ e_2\ \texttt{do}\ \overline{st}\ \texttt{end}
\end{aligned}
$$

```
module desugar/core

desugar = innermost(Desugar)

Desugar = fail
```

```
module desugar/int
imports desugar/core

Desugar = BinOpToCall

BinOpToCall :
  f#([e1, e2]) -> |[ f(e1, e2) ]|
  where <is-bin-op> f

is-bin-op :
  ?"Add"
  <+ ?"Mul"
  // etc.
```

```
module desugar/control
imports desugar/core

Desugar =
  ForToWhile <+ IfThenToIfElse

ForToWhile :
  |[ for x := e1 to e2
       do st* end ]| ->
  |[ begin
       var x : int;
       var y : int;
       x := e1; y := e2;
       while x <= y do
         st* x := x + 1;
       end
     end ]|
  where new => y

IfThenToIfElse :
  |[ if e then st* end ]| ->
  |[ if e then st* else end ]|
```

```
module eval/core

eval =
  eval-special
  <+ all(eval); try(eval-exp)

eval-special =
  EvalVar <+ eval-stats
  <+ eval-assign
  <+ eval-declaration

eval-assign =
  |[ x := <eval => e> ]|
  ; rules(EvalVar.x :
      |[ x ]| -> |[ e ]|)

eval-declaration =
  ?|[ var x : t; ]|
  ; rules(EvalVar+x :- |[ x ]|)

eval-stats =
  Stats({|EvalVar : map(eval)|})

eval-exp = fail
```

```
module eval/int
imports eval/core

eval-special = eval-or <+
   eval-and

eval-exp =
  EvalAdd
  <+ EvalMul
  // etc.

EvalAdd :
  |[ Add(i, j) ]| -> |[ k ]|
  where <addS> (i, j) => k

EvalMul :
  |[ Mul(i, j) ]| -> |[ k ]|
  where <mulS> (i, j) => k
```

```
module eval/control
imports eval/core

eval-special =
  eval-if <+ eval-while
  ; eval

eval-if =
  |[ if <eval> then <*:id>
          else <*:id> end ]|
  ; EvalIf; eval-stat

eval-while :
  st@|[ while e do st* end ]| ->
  |[ if e then st* st else end ]|

EvalIf :
  |[ if i then st1*
          else st2* end ]| ->
  |[ begin st1* end ]|
  where <not-zero> i

EvalIf :
  |[ if 0 then st1*
          else st2* end ]| ->
  |[ begin st2* end ]|
```

Figure 3.1 The TFA language. Columns are modules core, int, and control, rows are aspects grammar, desugaring, and evaluation.

known as match and build). For example, a rewrite `?Add(x, y); !Add(y, x)` swaps the subterms of an `Add` term, by sequentially composing a match and a build. Pattern matching is a first-class operation. That is, it is not bound to the use in a pattern matching construct that handles all cases. Thus, a pattern match fails when it is applied to a term that does not match, and is allowed to do so. Failure is first class (any transformation may fail) and is handled by the choice combinator <+. The choice `s1 <+ s2` between two transformations first applies `s1` and when that fails applies `s2`.

In general, Stratego provides composition of transformation strategies from pattern matching and instantiation using a small set of combinators for control (including sequential composition and choice) and generic term traversal.

Named Strategies and Rules. A strategy definition `f = s` names a strategy expression `s` and can be invoked as a combinator using its name. For example, the definition `desugar = innermost(Desugar)` defines the `desugar` transformation as the application of the `innermost` strategy with the `Desugar` rule(s). A rewrite rule `f : p1 -> p2 where s` is sugar for a strategy `f = {x,..: ?p1; where(s); !p2}`, i.e. it rewrites a term matching `p1` to an instance of `p2` provided that the condition `s` succeeds. Using these features, we can define many small transformation components (rules, strategies) and make multiple different compositions from these basic components. For example, we can compose rewrite rules using choice (e.g. `EvalAdd <+ EvalMul`), to apply multiple rules to a term.

Open Extensibility. The key to the extension of transformations illustrated in Figure 3.1 is Stratego's open extensibility of rules and strategies. A module may provide several definitions for the same name. And such definitions may be defined across several modules, independently of each other. For example, the `desugar/core` module defines the `desugar` strategy in terms of the `Desugar` strategy. It defines the latter as the `fail` strategy, which always fails. Thus, applying the `desugar` strategy as defined in that module performs the identity transformation. However, the `desugar/int` and `desugar/control` modules (independently) extend the definition of `Desugar`. When combining these modules the definitions of `Desugar` are combined to

```
Desugar = fail <+ BinOpToCall <+ ForToWhile <+ IfThenToIfElse
```

and the `desugar` strategy normalises a term with respect to those rewrite rules.

This extensibility feature is the core reason that Stratego has a whole program compiler. All modules of a program are combined in order to gather and combine all the extensions of each strategy. In addition, there are two more features that complicate separate/incremental compilation: dynamic rules and overlays.

Dynamic Rules. Dynamic rules are rewrite rules that are defined dynamically during a transformation using the `rules(...)` construct. For example, the evaluation strategy in Figure 3.1 uses a dynamic rule `EvalVar` to map variables to values. Furthermore, a dynamic rule gives rise to a number of derived strategies for applying and reflecting on the dynamic rule. There are currently

18 such derived strategies. Only one definition of each of these derived strategies should be generated per dynamic rule name, even if it is defined in multiple modules. Therefore, this feature requires global program information.

Overlays. Overlays (which do not appear in the example in Figure 3.1) are pattern aliases that can be used in both match and build position (E. Visser 1999). The applications of overlays are syntactically indistinguishable from regular terms. Overlays are greedily expanded; at run-time they are no longer available by name. A module does not need to import the module that defines the overlay for it to be available. Therefore the compiler needs to know all overlay definitions in the entire program to identify all their uses and expand them, affecting incremental compilation.

Open Extensibility in the WebDSL Compiler. The extensibility features discussed above are used in practice in compilers for languages used in production. For example, WebDSL, a domain-specific web programming language (Groenewegen, Hemel et al. 2008; Groenewegen and E. Visser 2013), consists of a number of sublanguages such as for the definition of data models, querying, computation, user interface templates, and access control (E. Visser 2007; Groenewegen and E. Visser 2008). The WebDSL compiler uses the open extensibility of Stratego to define extensions to basic analysis and transformation strategies per sublanguage. We present some statistics of the WebDSL language definition in Table 3.1 to show the relevance of the previously identified Stratego features in a real-world language definition.

The code base of the compiler consists of 399 modules with 2644 distinct strategies in the source code. Many additional strategies are generated as helpers of dynamic rules (6678), and constructors of terms (769) are usable as strategies as well. A total of 10091 strategies are compiled by the stratego compiler when the entire project is built. Out of those strategies, 857 have definitions in more than one module, showing how much the cross-modular extension feature is used.

Ambiguous strategy use sites are numerous within the codebase, 942 calls to a strategy are initially ambiguous in arity. Overlays are not used very much, only 19 definitions exist in the codebase and 8 of those are not even used any more. Most overlays are only used in one module (usually the defining module), although one is used in 23 modules.

The project uses 371 dynamic rules, most of which (334) have only one definition. These are most likely used as scoped mutable variables, rather than proper dynamic rewrite rules. That still leaves 37 dynamic rules with multiple definitions in different places.

3.3 Compiler Architectures and Build Systems

Compilation of Stratego's open extensibility requires the integration of definitions from multiple modules, precluding a simple separate compilation model. In this section, we first recall how in separate compilation (for the C language) a compiler outsources dependency tracking and computing which files to recompile to a build system; an idea we build on for our incremental compiler.

# of modules	399
# of distinct named strategies	10 091
# of congruences	769
# of dynamic rule helper strategies (18 × 371)	6678
# of user-defined strategies (rest)	2644
# of modules in which a named strategy is defined	10 091
1 module	9234
2–10 modules	826
11–20 modules	19
21–30 modules	8
> 31 modules	4
# of possibly ambiguous strategy use sites	942
# of overlay definitions	19
# of modules in which a distinct overlay is used	19
0 modules	8
1 module	9
2 modules	1
23 modules	1
# of distinct dynamic rule names	371
# of contributions per dynamic rule name	371
1 contribution	334
2 contributions	26
3 contributions	5
4 contributions	3
8 contributions	2
10 contributions	1

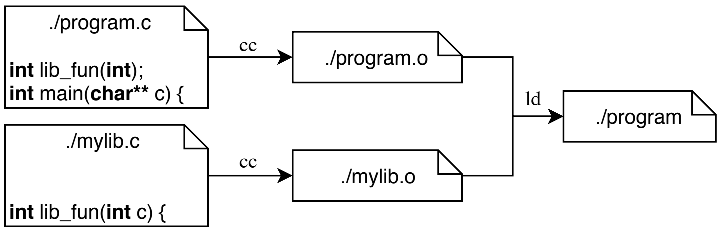
Table 3.1 Code metrics of the WebDSL compiler code base.

Then we review the original whole program compilation model for Stratego and its limited support for separate compilation. We also discuss a previous attempt at incremental compilation by dynamic linking.

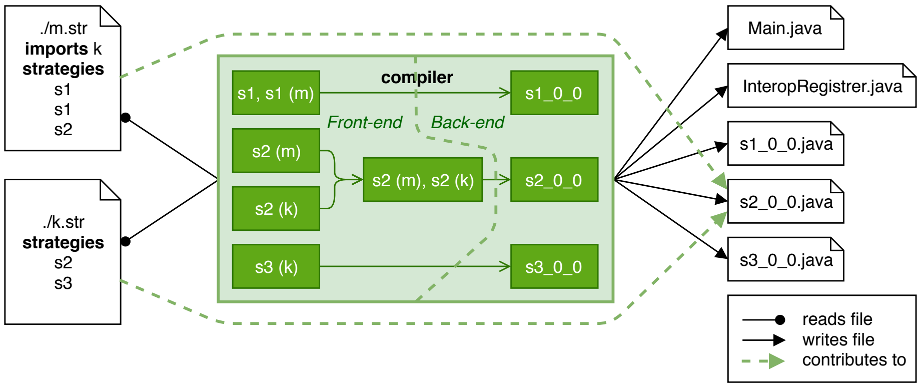
3.3.1 Separate Compilation and Build Systems

Separate compilation allows efficient recompilation of programs consisting of multiple compilation units. Only the compilation units that are affected by a change need to be recompiled. Typically, the work of determining which compilation units to recompile is not done by the compiler, but by an external build system. This is a nice separation of concerns. The compiler is not complicated by tracking dependencies and whether target files are up-to-date and the build system has to know only the external interface of the compiler.

For example, consider the C programming language and the compilation scenario illustrated in Figure 3.2. C has a separate compiler, which translates a `.c` source file and to `.o` object file. Cross-module static checking is enabled by means of 'forward declarations' of functions (which are typically defined in



☙ **Figure 3.2** An example build of a simple C project.



☙ **Figure 3.3** Whole-program compilation example for Stratego. All modules are discovered and merged by a single process.

additional header files included during preprocessing), defining the required interface to be implemented by the other files. Object files are combined into an executable by a linker. Thus, building a C program involves compiling all its files and then linking the resulting object files. Most C projects use a build system such as Make for this process such that only the files that are changed (or are affected by a change in a header file) need to be recompiled.

While this separation of concerns between compiler and build system is attractive for the compiler writer, it is less so for the programmer who needs to make sure the build system configuration accurately reflects the dependencies in the program often leading to unsound configurations, requiring frequent expensive from-scratch builds.

3.3.2 *Whole-Program Compilation*

The simple separate compilation model of C does not apply to Stratego because strategy definitions across modules need to be integrated. Therefore, Stratego's compiler uses a whole program compilation model (Bravenboer, Kalleberg et al. 2008) as illustrated in Figure 3.3. (The original compiler translated Stratego to C. The current compiler produces Java code. The compiler architecture is unchanged.) The compiler reads all module files by following imports from the main file of a Stratego program. After parsing the files, the compiler builds a single internal model of the program. Strategy names (which can

```
desugar(|e) =
  CallT(\SVar(n) -> n\, id , id)
desugar(|env) :
  Var(e) -> <lookup> (e, env)
```

$\Rightarrow$

```
desugar(|e1) =
  CallT(\SVar(n) -> n\, id , id)
  <+ {e: (Var(e) -> <lookup> (e, e1))}
```

Figure 3.4 Integration of strategy definitions with the same name.

be overloaded in addition to have multiple definitions) are disambiguated by including the arity in the name. Then multiple definitions with the same name are merged into a single definition. Strategy definitions are merged by renaming the arguments to be consistent, scoping each strategy body, and defining the merged body as a choice between the two bodies, as illustrated in Figure 3.4. The order between definitions from different modules is undefined. The back-end of the compiler translates the simplified, merged definitions to Java classes and extracts information such as term constructors, constants, and strategy names to put into the `Main` and `InteropRegistrer` classes.

This process is illustrated by Figure 3.3. Strategies `s1` and `s3` are only defined in one module, and each have their corresponding Java class in the compiler output. Strategy `s2` is defined in both modules, and merged by the compiler into a single Java class.

Limited Separate Compilation Support. While the Stratego compiler is a whole-program compiler, it has support for separate compilation of libraries. Compiled libraries consist of an interface (list of strategies and term constructors), and the Java classes with the compiled code. However, these libraries are second-class citizens. A compiled library can only be imported in its entirety since its internal module structure is discarded. Furthermore, separate compilation is achieved by limiting open extensibility to library boundaries; an (external) strategy defined in a library can not be redefined or extended. Dynamic rules defined in a library cannot be extended either.

To recover some of the expected functionality for external strategies, Stratego provides two special keywords, `extend` and `override`, which can be applied to a definition that is already defined in an imported library. The `override` keyword overrides the definition with the local one, while the `extend` keyword overrides the definition but can call the original through the `proceed` keyword. These modifiers can only be applied to a single definition, i.e. these definitions cannot be further extended like regular strategy definitions. Calls to `proceed` have the same overhead as that of a normal strategy call, thus making this form of strategy extension more expensive at run-time too.

Thus, separate compilation allows the creation of libraries with reusable functionality, but without the idiomatic extensibility features of Stratego to interact with that code.

3.3.3 Independent Compilation with Dynamic Linking

In a case study of the Stratego compiler for the Pluto incremental build system, Erdweg, Lichter and Weiel (2015) developed a "file-level incremental" compiler for Stratego. To understand that model, we first discuss the Stratego to Java compilation scheme.

```
desugar(|env): Var(e) -> <lookup> (e, env)
```

$\Downarrow$

```
public class desugar_0_1 extends Strategy {
  public static Strategy instance = new desugar_0_1();

  public IStrategoTerm invokeDynamic(IStrategoTerm term, Stratego[] sargs,
  IStrategoTerm[] targs) {
    // ... match Var, bind e to local variable, build pair, bind to current term
  ...
    term = lookup_0_0.instance.invoke(term);
    return term;
  }
```

Figure 3.5 Stratego to Java compilation scheme.

Java Compilation Scheme. Figure 3.5 illustrates the compilation scheme. Each compiled strategy definition is a class that extends the `Strategy` abstract class. This class has invocation methods for a number of different arities with a default implementation that dispatches to a general method for two arrays of arguments (the strategy and term arguments). A generated strategy class extends this abstract class and implements the general method.

Strategy names are translated to the same name with underscores instead of dashes and followed by their arity separated by underscores. Each class has a public static field `instance` where the instance of the strategy is saved. This field is used to be able to override or extend strategies in compiled libraries. The overriding or extending strategy is a new class that extends the original class, overrides the invocation method, and overrides the instance field of the original class with an instance of itself.

The `Main` class that the backend generates provides an entry point into a Stratego program. If a `main` strategy is defined, it can be called through the `Main` class, which sets up the execution context, compute shared constants and term constructor objects, and execute the program. The shared constants and term constructor objects are stored in fields of the `Main` class and referenced by the other classes.

The `InteropRegister` registers all compiled strategies in the context object for interoperability with the Java implementation of the Stratego interpreter. This hybrid interpreter can load compiled libraries like the Stratego standard library through these `InteropRegisters`, and then interpret the main program as an AST which was only compiled with the compiler frontend.

Dynamic Linking Compilation Scheme. Erdweg, Lichter and Weiel (2015) experimented with a separate compilation model for Stratego by relying on dynamic linking. To the best of our understanding from inspecting the code, this was an independent compiler, i.e. it skipped cross-module static analyses. The case study demonstrates the applicability of the Pluto build system to dependency tracking and rebuilding within a compiler. However, no speedup is reported on, nor were the changes merged into the Stratego compiler.

The compilation model is illustrated in Figure 3.6. The compiler produces

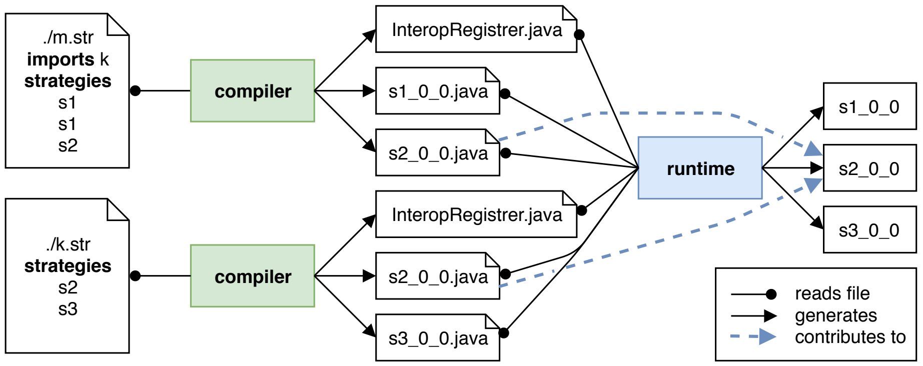


Figure 3.6 Dynamic linking, incremental compilation example for Stratego. The compiler is run for each module, governed by the Pluto build system. An adapted runtime merges strategies dynamically, i.e. during program start-up.

for each Stratego source file a list of Java classes for the strategy definitions in that file. With some changes to the Stratego runtime system, the Java classes are merged at run-time rather than merged statically. At the start of the program, all loaded classes register themselves by name to a `StrategyCollector`, as illustrated in Figure 3.7.

The `Main` class initialises fields for every strategy by looking up the strategy by name from the collector. The collector builds `StrategyExecutor` objects for this which contain an array of `Strategy` classes that are produced by the compiler. These are attempted in order when the `StrategyExecutor` is invoked. Although the `HashMap` lookup based on strategy name is only done once for every strategy call, each strategy definition from a different module requires a separate invocation in a different class.

3.3.4 *Summary*

Whole-program compilation requires minimal configuration and is therefore easy to use and requires less maintenance. However, whole-program compilation does not scale because recompilation time is proportional to the size of the entire program instead of the size of the change. We would like to keep the low-configuration benefit of whole-program compilation, while scaling incrementally with the size of the change. Pure separate compilation cannot be applied, as whole-program knowledge is required for compilation. The disadvantage of the dynamic linking model is that it shifts the burden of integrating strategy definitions to run-time, which incurs execution overhead and requires disruptive changes to the compilation scheme. Therefore, we need a hybrid-incremental compilation approach, which we describe and apply to Stratego in the next section.

```
desugar(|env): Var(e) -> <lookup> (e, env)
```

$$\Downarrow$$

```
public class desugar_0_1 extends RegisteringStrategy {
  public static final Strategy instance = new desugar_0_1();

  public static Strategy getStrategy(Context context) {
    return context.getStrategyCollector().getStrategyExecutor("desugar_0_1");
  }

  public void registerImplementators(StrategyCollector collector) {
    collector.registerStrategyImplementator("desugar_0_1", instance);
  }

  public void bindExecutors(StrategyCollector collector) { ... }

  public IStrategoTerm invokeDynamic(IStrategoTerm term, IStrategoTerm arg1) {
    // ... match Var, bind e to local variable, build pair, bind to current term
 ...
    term = Main.lookup_0_0.invoke(term); // note the difference in invocation!
    return term;
  }
}
```

Figure 3.7 Compilation scheme with dynamic linking.

3.4 Incremental Compilation

In this section we first present our approach to incremental compilation in general. Then we make this concrete by presenting the application of our approach to the Stratego language and compiler.

3.4.1 *A General Blueprint for Incremental Compilation*

Our approach to incremental compilation results in a compiler that, from an external interface, looks like a whole-program compiler, not a separate compiler. Internally it uses an incremental build system that handles dynamically discovered dependencies. This build system caches intermediate results within the compiler to make it incremental.

Data Splitting. The key idea is that every file that the compiler reads, is split up into parts as soon as possible. Each of these parts is then processed by separate build tasks. This allows the build system to pick up on parts that are unchanged in the file and parts that are changed. This idea is what gives the compiler sub-file incrementality.

The size of the parts influences how incremental the compiler is. A coarse-grained split results in less incrementality as more code unrelated to a change is recompiled. A fine-grained split increases the interaction between the parts during compilation steps, so that the overhead of the build system tracking all the dependencies becomes higher than the gains from avoiding repeated compilation of unchanged code.

The split to choose depends on the language features, and how much of the original compiler to reuse as is. A reasonable split is to choose parts that become separate outputs of the compiler. However, a finer-grained split is also possible, where parts are merged in a later stage of the compiler.

Compiler Stage Splitting. Using an incremental build system to compose a compiler allows defining compiler stages as build tasks. The outputs of build tasks are cached and can be reused on recompilation. More tasks means more dependency tracking and thus more overhead. The number of tasks mostly depends on the granularity of the data split. However, even stages that operate in succession on a single part of the data can be split to improve performance. The main idea behind splitting up a successive operations is to avoid the later (expensive) operations in the chain, if an early operation results in the same output despite the change in input. That is, if an early operation gives the same output for different inputs, and the following operations are expensive, it may be worth the build system overhead to split the operations over multiple tasks. This allows the build system to cut off early when it observes that an intermediate result has not changed. The trade-off should be made by taking into account the overhead of the build system for such a change, the chance that this situation happens, and the expense of the later operations.

3.4.2 *Incremental Compilation for Stratego*

To apply this blueprint to Stratego, we split modules by strategy definition and some extracted information for the static analyses at the end of the front-end of the compiler, as illustrated in Figure 3.8 on page 78. Then we merge strategy definitions with the same name and arity, and call the compiler back-end for each of these merged definitions. This results in one sub-front-end task per strategy in a file, one front-end task per file, and one back-end task per merged strategy, to minimise the work when a small change is made in a file.

The static analyses are done non-incrementally and were found to take an insignificant amount of time in our benchmarks (Section 3.5). Other languages with more expensive static analyses may benefit from an incremental static analysis stage.

PIE. As our build system we use PIE (Konat, Steindorfer et al. 2018; Konat, Erdweg and E. Visser 2018), a sound and incremental build system supporting dynamically discovered dependencies to build tasks and files. Build tasks are regular (imperative) functions, except that they call other tasks (and can use their return value) by 'requiring' them, which instructs PIE to record a dependency. Similarly, build tasks can require (read from) or 'provide' (write to) files, which records dependencies to those files. PIE only re-executes tasks when their input changes, when the return value of a required task changes, or when a required/provided file changes. Otherwise, PIE returns the cached value of a task, providing incrementality implicitly, freeing the build developer from having to explicitly handle incrementality.

Build Algorithm. In Listing 3.1 we present the build orchestration algorithm. It includes type definitions and helper functions in listing 3.1a, which combine information from one front-end task into the available global information.

```
type M = String // Module name
type S = String // Strategy name
type C = String // Constructor name
type AST = ATerm // Stratego Core AST
type SI = struct: // Static Information
  imps: Rel[M,M], // Imports
  defStrs: Rel[M,S], // Defined strategies
  defCons: Rel[M,C], // ... / constructors
  usedStrs: Rel[M,S], // Used strategies
  usedCons: Rel[M,C], //  ... / constructors
  strUsedCons: Rel[S,C], // ... per strategy
  strASTs: Rel[S,AST], // Definitions of strats
  olayASTs: Rel[C,AST], // ... / overlays
type FI = struct: // Front-end Information
  imps: Rel[M,M],
  defStrs: Set[S],
  defCons: Set[C],
  usedStrs: Set[S],
  usedCons: Set[C],
  strUsedCons: Rel[S,C],
  strASTs: Rel[S,AST],
  olayASTs: Rel[C,AST],
type LI = struct: // Library Information
  defStrs: Set[S],
  defCons: Set[C],

func combineInfo(si: SI, mod: M, fi: FI):
  si.imps[mod] := fi.imps ∪ defaultImps
  si.defStrs[mod] := fi.defStrs
  si.defCons[mod] := fi.defCons
  si.usedStrs[mod] := fi.usedStrs
  si.usedCons[mod] := fi.usedCons
  si.strUsedCons ∪= fi.strUsedCons
  // Strategies with the same name go together:
  si.strASTs ∪= fi.strASTs
  si.olayASTs ∪= fi.olayASTs

func combineInfoLib(si: SI, mod: M, li: LI):
  si.defStrs[mod] := li.defStrs
  si.defCons[mod] := li.defCons

task frontEnd(M) → FI
task subFrontEnd(S) → FI
task frontEndLib(M) → LI
task backEnd(S, Set[AST], Set[AST])
```

(a) Type definitions and combining front-end information into the static information struct.

```
task main(mainMod: M):
  W: List[M] := [mainMod] // Worklist
  // Prevent loops for cyclic module imports
  seenMods: Set[M] := {mainMod}
  staticInfo: SI := SI() // Static analysis info
  // FRONTEND + COLLECT STATIC ANALYSIS INFO
  while W is not empty:
    mod := W.pop()
    if mod.isLibrary():
      // cached task call
      li := req frontEndLib(mod)
      combineInfoLib(staticInfo, mod, li)
    else:
      fi := req frontEnd(mod) // cached task call
      combineInfo(staticInfo, mod, fi)
      // Follow imports
      W.pushAll(staticInfo.imps[mod] \ seenMods)
      seenMods ∪= imps[mod]
  // STATIC ANALYSIS
  staticChecks(mainMod, staticInfo)
  // BACKEND
  for unique (_, str) in staticInfo.defStrs:
    olayASTs′ := {}
    for con in staticInfo.strUsedCons[str]:
      olayASTs′ ∪= staticInfo.olayASTs[con]
    // cached task call:
    req backEnd(str, staticInfo.strASTs[str],
  olayASTs′)

func staticChecks(main: M, si: SI):
  visStrs := si.defStrs; visCons := si.defCons
  for scc in topoSccs(main, si.imps):
    visCons′ := {}; visStrs′ := {}
    for mod in scc:
      visStrs′ ∪= visStrs[mod]
      visCons′ ∪= visCons[mod]
      // propagate info from earlier iter
      for m in si.imps[mod]:
        visStrs′ ∪= visStrs[m]
        visCons′ ∪= visCons[n]

    for mod in scc:
      visStrs[mod] := visStrs′
      visCons[mod] := visCons′
      assert(usedStrs[mod] ⊆ visStrs′)
      assert(usedCons[mod] ⊆ visCons′)
```

(b) Pseudocode for the build orchestration.

Listing 3.1 The build orchestration algorithm, simplified to fit on one page. Tasks (which are incrementalized) are defined with the `task` keyword, and called (required) with `req`. The `subFrontEnd`, `frontEndLib`, and `backEnd` tasks are calls into parts of the original compiler, wrapped as tasks so they are incrementalised by the build system. The front-end tasks require the file corresponding to their input module, and the back-end tasks provide the generated files, ensuring re-execution when the input files change.

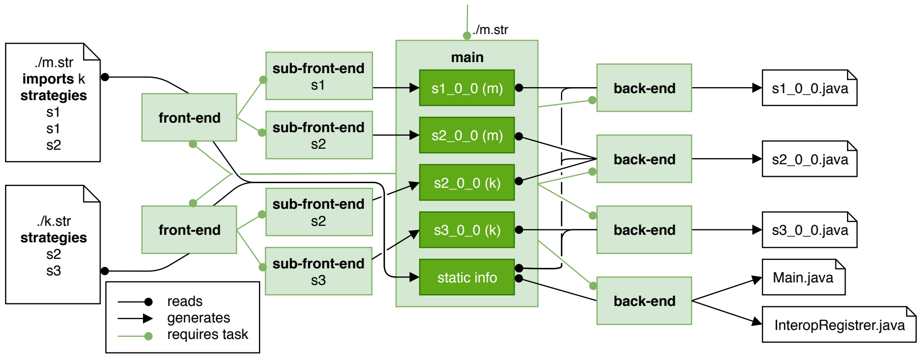


Figure 3.8 Static linking, incremental compilation example for Stratego. A frontend task for each file, a backend task for each strategy.

Note the commented line, which combines maps of strategy name to strategy definition AST. This is the point where strategies from different modules are combined by name.

Listing 3.1b contains the pseudocode of the build algorithm. The `main` task which orchestrates the build uses a worklist, starting from the main module, to run frontend tasks for each module. New modules are found through imports of processed modules.

The output of the front-end calls are collected and merged by strategy or constructor name, which is used in the static analysis (`staticChecks` function). Note that `staticChecks` is a function, not a task, and is therefore not cached, because it requires global information which changes almost every run, making caching moot. If `staticChecks` succeeds, the backend is called once per strategy, which takes a list of ASTs for that strategy (the contributions from different modules) and a list of ASTs for overlays that are used in the strategy.

Static Analysis. In order to be compatible with the original whole program compiler, we reimplement the static analysis in a `staticCheck` function that takes in the gathered static information.

We use a topological order over the import graph, so we can easily propagate the strategy definitions according to the transitive import semantics. But since the graph can have cycles, we process the strongly connected components (SCCs) of the import graph instead of individual modules. In other words, a group of modules that import each other is processed together.

The visible strategies and constructors are computed first, then the used strategies and constructors are checked to be a subset for each module. To fit all the code on one page, we have elided some other static analyses from the pseudocode in Listing 3.1. This includes some analysis information that is used by all back-ends. The following analyses are included in the implementation:

Ambiguous Strategy Call Resolution Ambiguous strategy calls result from the use of bare strategy names in strategy argument position. We resolve these names by looking for strategies of any arity with the same name. If a strategy exists with arity 0/0, the reference must resolve to this strategy and is considered unambiguous. When this arity does not exist, but there is only one arity available, the name can also be resolved. In other cases, where zero or more than one strategies are found, we report an error. The resolved names are returned by the static analysis. The resolutions are used in the back-end to replace the original names with the ones that include arity.

Extending and Overriding Strategies To be able to extend or override a strategy, an external strategy from a library needs to exist. External strategies are declared by a library along with the compiled implementations. We check for non-overlap of the relevant sets and give errors on any that are found.

Cyclic Overlay Check While looking into the original compiler details for overlays, we discovered a missing check. The original compiler greedily expands overlays. Therefore an overlay definition `overlays A() = A()` will cause the compiler to loop. We added a check to fix this bug, which checks if there for cycles between any overlays (self-loops and indirect cycles). Overlays are only pattern aliases, without conditionals, so this analysis can be complete. We build a dependency graph of overlays and check for cycles.

Implementation Effort. The incremental Stratego compiler was developed by one of the authors over a period of 10 months. Given the time spent on other tasks, this comes down to approximately 6-7 months of full-time equivalent. Before the start of the project the author was not very familiar with the compiler internals of Stratego.

The original Stratego compiler architecture certainly helped to make this quick development possible. The compiler generally transforms immutable trees (the AST), which makes it relatively easy to split off parts of it into cacheable tasks. The complicating factor is some mutable data structures that are used for extra (analysis) information. The mutation of these data structures would not be cached if naively separated into tasks to be cached.

Conclusion. Our incremental compiler is an instantiation of our general blueprint for hybrid-incremental compilers. We use an incremental build system internally to incrementalise compilation tasks. Our compilation scheme splits files by top-level definition resulting in incrementality per top-level definition. It reuses the front-end of the original compiler on each top-level definition separately, and caches the results. After the front-end, the compiler performs all static analyses, merges top-level definitions with the same name and arity, and reuses the backend of the original compiler on each top-level definition separately. The output is completely backward compatible with the original runtime system.

3.5 Evaluation

We evaluate the performance of our incremental Stratego compiler by comparison against the original compiler. In particular, we evaluate the following research questions:

RQ1 Does the incremental compiler scale with the size of the changes?

RQ2 What is the overhead of a clean build with the incremental compiler?

RQ3 Is the incremental compiler correct with respect to the original compiler?

3.5.1 Research Method

In a controlled setting, we compare the performance of the original compiler and our incremental compiler. We run our experiments on a MacBook Pro (Early 2013) with an Intel Core i7 2.8 GHz CPU, 16 GB 1600 MHz DDR3 RAM, and an SSD. The machine is running Mac OS 10.14.5, Java OpenJDK 1.8.0_212, Docker 2.0.0.3 (31259), VirtualBox 5.2.8 r121009 (Qt5.6.3), and Spoofax 2.5.8. As the experiments are performed inside a virtualised environment (Docker), the absolute numbers of our measurement may be higher than typical usage of the compiler in a normal setting. However, the virtual machine image can be easily reused by others to reproduce our experiments.

Subject. As a benchmark subject we use the WebDSL language implementation. We use the version control commit history from 2016 to August 2019, consisting of 200 commits.

Data Collection. We perform measurements by repeating the following steps. We run a clean build with the incremental compiler. Then we step to the next commit and pass the list of changed files to the compilation system. We do this for each commit. We measure the elapsed time for each successive compilation step, through Java's `System.nanoTime`.

We warm up the JVM on which the compiler runs by compiling a project and 3 successive commits, followed by a garbage collection. This was established by repeatedly running the benchmark on a cold JVM for many commits and noting the stabilisation of result times within 3 commits. We measure each project in a separate JVM invocation. During the JVM invocation we warm up once, and repeat the measurements 5 times. The results are shown as stacked bar-plots (with the arithmetic mean for each part) with whiskers to show the sample standard deviation (SD) in compilation time.

Similarly we run the original compiler for each project. This is only done to check that the compilation time of the original compiler does not vary much. We warm up the JVM as before with 1+3 compilations, and afterwards measure for each commit, and again go through the history 5 times. The results are shown with whiskers to show the full range (R) in compilation time.

3.5.2 Results

Scaling. Of all 200 commits, 2 commits failed to build and were excluded from the results. We inspected these and determined that these failures were due to bugs that were fixed in the next commit. They failed to build with the original compiler as well as the incremental compiler.

Figure 3.9 shows the results of our benchmark for the WebDSL codebase. We found 78 commits that do not change any Stratego files and are excluded from the plot for that reason. The first commit, which is not compiled incrementally is also excluded. The remaining 119 commits are sorted in two different ways for comparison. The number of changed source files are shown in the top plot

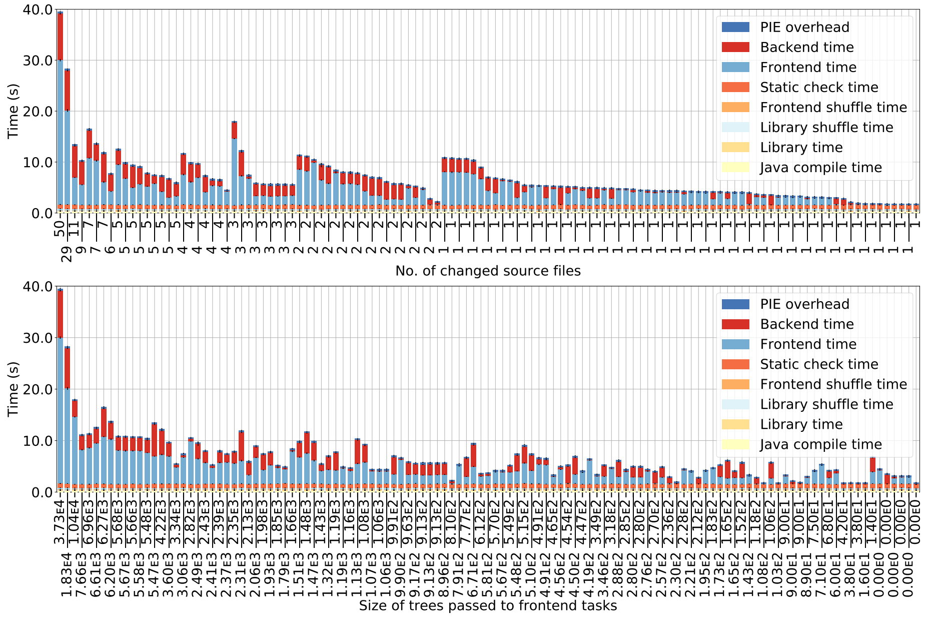


Figure 3.9 Benchmark results of the incremental compiler on the WebDSL codebase from two perspectives. The results are displayed per commit, broken down into different parts of the incremental compiler. The Y-axis is time in seconds. The X-axis is number of changed files in the top plot, and the size of the trees passed to the frontend in the bottom plot.

on the x-axis, the bottom plot shows the sum of sizes of (abstract syntax) trees passed to sub-front-end tasks.

The number of changed source files does not have a direct correspondence with the amount of work the incremental compiler needs to do. Although it is in direct relation with the effort for parsing, subsequent parts of the compiler deal only with changed top-level definition. The plot shows how the total incremental compile time does not scale directly with the number of changed files.

The front-end times are closer, but still not one-to-one, correlated with the sizes of trees passed to them. Some Stratego language constructs are desugared into much larger pieces of code which are then processed further, so a small change to the text or tree can still have a non-proportional impact.

The code generation part of the compiler is a more straight-forward translation. If we would order the plot by sizes of trees passed to the back-end tasks, we would see a strict decline of the back-end times over the plot.

Overhead. We compare the overhead of the incremental compiler during a clean build, as this is where the user of the incremental compiler will notice the overhead. For the WebDSL codebase, the observed clean build time for the incremental compiler is 178.53 seconds, whereas the build time of the original compiler is on average 93.54 seconds (see also Figure 3.10). In other

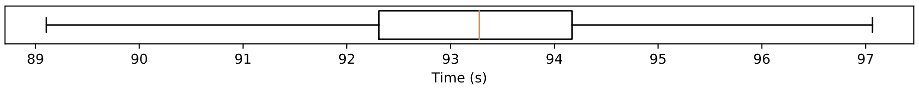


Figure 3.10 Benchmark results of the original compiler on the WebDSL codebase. The X-axis is time in seconds. This is a boxplot over the measurements for all commits, as they have low variance. The whiskers span the full range of measurements.

words, the incremental compiler has an overhead of 90.8 % for clean builds on the WebDSL codebase (RQ2). All incremental compilations are under the lowest fastest time with the original compiler. The overhead of the incremental compiler may have a variety of different causes.

Correctness. To test the correctness of our compiler, we ran the full WebDSL compiler test suite. We used the result of compiling the WebDSL project with the incremental compiler and the original compiler. The two resulting WebDSL compilers had the same behaviour for all tests (RQ3) (in both successes and a number of failures, which were expected behaviour according to the WebDSL developers).

3.5.3 Interpretation and Discussion

Although the scaling behaviour of the incremental compiler does not entirely follow the size of changes to files in the benchmark, the general tendency is there (RQ1). When we inspect particular pairs of commits with unexpected ordering we find that unexpectedly cheap (in incremental compile time) commits change a strategy with the same name over multiple files. Unexpectedly expensive commits make changes in strategies that use expensive language features that are expanded by the compiler to much larger pieces of code.

Something we should note is a known issue with the official Stratego parser we use. This parser can take a relatively long time to parse certain Stratego files. This issue is solved in a new version of the parser, but that version is currently not stable enough for us to use in the benchmark.

The size of the overhead seems to reside in something other than the overhead of PIE itself and the shuffling of information. Our reimplementation of the static analysis of Stratego is also not the problem. It is unclear what exactly causes the overhead, but somehow the multitude of calls into the original compiler code or the separate treatment of the different files is causing overhead in the incremental compiler. We have attempted experimental changes to the compiler to verify or falsify assumptions on where this overhead comes from, but so far without success.

Compared to the time it takes for every compilation with the original compiler, a single clean build is a cost that anecdotally Stratego users are willing to pay for the speedup they receive on later incremental compilations. Despite the overhead that is visible in the clean build, our incremental compiler provides a speedup of up to $\sim 90\times$ for small changes in one or two files. Most commits take less than 10 seconds to compile, versus the 1.5 minutes that the original compiler takes, massively increasing productivity for Stratego programmers.

3.5.4 Threats to Validity

We explicitly discuss the threats to validity of this benchmark. In particular, we address generalisability of the results (external validity), factors that allow for alternative explanations (internal validity), and suitability of metrics for the evaluations goals (construct validity).

External Validity. A threat to the generalisability of our results is that we only evaluated our Stratego compiler on Stratego code that originated from our research group. Therefore it has a certain code style that may influence the efficacy of our compilation approach. With the inclusion of WebDSL we cover all Stratego features and the classic Stratego code style with highly overloaded strategies, which are more expensive to compile incrementally.

Internal Validity. A factor that allows an alternative explanation for our results is miscompilation, where only part of the program was actually successfully compiled, and another part was ignored. Early in the development of the compiler we found strange measurement results that came from the error recovery in the parser. Because of a misconfiguration, the Stratego parser sometimes failed to parse the entire file. Error recovery in the parser ensured that an abstract syntax tree was still returned, containing only those top-level strategy definitions that could be parsed successfully. Because our new compiler did not check if the parser had recovered from errors, parts of the Stratego program were not actually compiled. We found and fixed this problem through careful inspection of the benchmark results, and avoided more problems by running the WebDSL compiler test suite.

Construct Validity. Regarding the suitability of metrics for the evaluation goal, we measure performance using elapsed time only. We control JIT compilation with a warmup phase. By running the garbage collector in between measurements, and monitoring the available memory, we control memory availability during all measurements. However, the incremental compiler stores intermediate results in memory and may perform differently in environments with less memory available.

Of course we can also not entirely eliminate background noise. However, we have repeated all measurements five times and see low variance between measurements (maximum sample standard deviation was 0.36 seconds). This is also visible in Figure 3.9, where we display black whiskers on each stacked bar, all of which are barely distinguishable.

The virtualisation of our benchmark allows easy distribution of the benchmark for reproduction of our results. Since we use a long-running macro-benchmark, the virtualisation does not significantly influence the results. The same benchmark, run on the same hardware without virtualisation, shows similar results. There is only a 0 % to 25 % reduction in times, where the longest times were reduced most, which shows the overhead of virtualisation.

3.6 Related Work

Recompilation. Independent compilation was an early way to speed up compilation by splitting up work and caching the intermediate results (Backus and Heising 1964). To reinstate guarantees of static analysis on the entire program, Mesa introduced separate compilation (Geschke, Jr. and Satterthwaite 1977) and inspired other languages such as Modula-2 (Wirth 2007) and (an extension of) Pascal (Kieburtz, Barabash and Hill 1978) to do the same. Incremental compilation was another refinement of the concept to allow a more arbitrary splitting of the work instead of by file (Reiss 1984; Ryan, Crandall and Medwedeff 1966).

The novelty of our approach is to apply these (already over 30-year-old) ideas in a new way. We use an incremental build system to piece together parts of an existing non-incremental compiler into an incremental compiler. The expressive power of the build system we use, with dynamically discovered dependencies, allows us to fully express the build of the language inside the compiler. Therefore, the build system becomes an invisible part of our compiler. The user does not need to configure it on a per project basis.

Build Systems. Relevant incremental build systems are of course the system we use, PIE (Konat, Steindorfer et al. 2018), as well as its predecessor Pluto (**LW15**). Pluto was in some ways more powerful than PIE as it can handle circular build dependencies which require a registered handler to compute a fixpoint. PIE, however, has much less overhead because it does not support this feature, and because it can do so-called bottom-up builds (Konat, Erdweg and E. Visser 2018). A bottom-up build requires the list of changed files and directories and uses that and the previous execution of the build script to do the minimal updates necessary to have a consistent output again. This gives less overhead from the build system because parts of the dependency graph do not need to be traversed at all.

Of course there are other build systems, both those that support incrementality and those in which incrementality can be hacked in by dynamically generating new build scripts and calling into those. Mokhov, Mitchell and Jones analyse the features of different build systems and give an overview of desirable features (Mokhov, Mitchell and Jones 2018).

Our approach is that the build system is an invisible part of our compiler. The build is defined once and for all inside the compiler, therefore the user does not need to configure it on a per project basis.

Incremental Compilers. While our compilation approach works well for reusing an existing compiler, there are other approaches to incremental compilation, especially when one is built anew.

An example of a different incrementalisation approach is JastAdd, a Java compiler built entirely on Reference Attribute Grammars (RAGs). These RAGs gave support for incremental evaluation (Söderberg and Görel Hedin 2012). To do something similar for the Stratego compiler we would need to build a completely new Stratego compiler in terms of RAGs, which is a non-trivial amount of work.

Another example of an incremental compiler is rustc, the compiler for the Rust programming language. The rustc compiler has an experimental incremental compilation mode that caches intermediate results, automatically records (dynamic) dependencies between these results, and reuses cached results when a dependency chain has not been invalidated (Woerister 2016). However, this incremental compiler mode is not enabled by default (at the moment of writing) (Woerister 2019), since it sometimes makes compilation slower because of the overhead of incremental computation. The difference with our approach is that the Rust compiler automatically tracks dependencies based on reads and writes of data structures, whereas in our approach, we explicitly state the units of computation and dependencies between these computations. While explicitly stating this requires more effort, the advantage is that we can tune the granularity of the incremental compiler, to prevent the case where fine-grained dependencies cause too much overhead.

Extension Unification. Extensibility has been studied from different perspectives. Erdweg, Giarrusso and Rendel created a classification of extensibility, where Stratego falls within 'extension unification' (Erdweg, Giarrusso and Rendel 2012, § 4). Extension is the terminology for supporting "language extension of a base language [where] the implementation of the base language can be reused unchanged to implement the language extension" (*ibid.*, § 2.1, Def. 1). Unification is the terminology for supporting "language unification of two languages [where] the implementation of both languages can be reused unchanged by adding glue code only" (*ibid.*, § 2.2, Def. 2). Stratego has a composition of extension and unification in that languages can be extended from a base (pre-defined strategies), and the different extensions (contributions to a pre-defined strategy) can be unified unchanged.

Expression Problem. The expression problem defined by Wadler has more stringent requirements (Wadler 1998). There is a reason its name includes 'problem'. Over the years there have been many suggested solutions (Torgersen 2004; Wang and Oliveira 2016; Kiselyov 2010) and extensions of the problem statement (Odersky and Zenger 2005; Kaminski et al. 2017). The cited papers are only a few examples.

Stratego does not have the strong static type system required by Wadler for the original expression problem. This gives the language some more flexibility to attain the other properties more easily. Both types and functions on those types can be extended, without a change to the original code. And as presented in this chapter, Stratego can now be separately compiled. There is also no linear order in language extensions defined in Stratego, as required by Odersky and Zenger (2005), nor is there any glue code for combining different language extensions as required by Kaminski et al. (2017).

Language Definitions. From a different perspective, if we look at competitors in language definitions we might look at attribute grammar systems such as Silver (Van Wyk et al. 2010) and JastAdd (Ekman and Görel Hedin 2007a). Silver has separate compilation and generates multiple separate Java classes so it can leverage the Java compiler for hybrid incremental compilation, if composed by an external build system. It also leverages some dynamic linking to allow more concerns to be compiled separately, and therefore make compilation more incremental (Van Wyk 2019). We are not aware of a publication on JastAdd's compilation scheme.

3.7 Conclusion

In this chapter we have presented a design approach for hybrid incremental compilers. These compilers look like a whole-program compiler from the outside, but leverage an internal, incremental build system for incremental compilation. The blueprint of this design includes considerations on how to split up data and how to split up compiler stages inside the build system. We have given a concrete example of the blueprint with an incremental compiler for Stratego.

To motivate the decisions we made in our instantiation of the blueprint, we have presented an analysis of the Stratego language and shown how different features require global information to be compiled. The particularly problematic feature is that of unifying top-level definitions with the same name into a single definition that attempts the different alternatives.

Our instantiation of the blueprint splits up a file into top-level definitions early, processes each, then combines all equal-named top-level definitions before moving on to code generation. The static analysis is reimplemented but all other parts of the compilation are recycled from the original compiler.

By going through the version control history of a large Stratego project, we have demonstrated the incrementality of the new compiler. The original compiler takes about 93±3 seconds to compile the project. Our results show that all but one commit with 1 file changed are compiled in under 10 seconds, a majority of 1 file changed commits compile in under 5 seconds, and even the largest commit which changes 50 files is incrementally compilable in under 40 seconds.

We would like to see another real-world programming language gain an incremental compiler with our design approach. The internally used build system and reuse of compiler components should make it a reasonably sized project that can greatly benefit the users of the programming language. It would also strengthen our hypothesis that this approach is reusable for other programming language compilers.

Chapter 4

Gradually Typing Strategies

Abstract. The Stratego language supports program transformation by means of term rewriting with programmable rewriting strategies. Stratego's traversal primitives support concise definition of generic tree traversals. Stratego is a dynamically typed language because its features cannot be captured fully by a static type system. While dynamic typing makes for a flexible programming model, it also leads to unintended type errors, code that is harder to maintain, and missed opportunities for optimisation.

In this chapter, we introduce a gradual type system for Stratego that combines the flexibility of dynamically typed generic programming, where needed, with the safety of statically declared and enforced types, where possible. To make sure that statically typed code cannot go wrong, all access to statically typed code from dynamically typed code is protected by dynamic type checks (casts). The type system is backwards compatible such that types can be introduced incrementally to existing Stratego programs. We formally define a type system for Core Gradual Stratego, discuss its implementation in a new type checker for Stratego, and present an evaluation of its impact on Stratego programs.

Based on: Jeff Smits and Eelco Visser (2020). 'Gradually typing strategies'. In: *Software Language Engineering (SLE), Proceedings*, pages 1–15. DOI: `10.1145/3426425.3426928`

4.1 Introduction

The Stratego language supports program transformation by means of term rewriting with programmable rewriting strategies (E. Visser, Benaissa and Tolmach 1998). Stratego's traversal primitives support concise definition of generic tree traversals. For example, the definition of `bottomup(s)` in Listing 4.4 on page 94 defines in one line a generic bottom-up traversal that can be instantiated with a selection of rewrite rules to be applied in a particular transformation, without needing to define a traversal for each constructor in the abstract syntax. Stratego is used in the Stratego/XT program transformation tool suite (Bravenboer, Kalleberg et al. 2008) and the Spoofax language workbench (Kats and E. Visser 2010) and used in production in research, education, and industry (Konat, Steindorfer et al. 2018; Denkers, van Gool and E. Visser 2018).

Stratego is a dynamically typed language, because its language features cannot be captured fully by a static type system. While dynamic typing makes for a flexible programming model, it also exposes Stratego programmers to unintended type errors. Static typing of strategies has been considered before by Lämmel and J. Visser (2002), Lämmel (2003), and others. Lämmel and Jones (2003) adopted Stratego's strategic programming in the SYB Haskell design pattern. These efforts focus on the statically typable fragment of strategies, making them unsuitable, as is, as a type system for Stratego. Furthermore, there is a considerable base of existing Stratego code, and having to convert that, at once, to statically typed code would preclude adoption of a type system.

In this chapter, we introduce a gradual type system for Stratego that combines the flexibility of dynamically typed generic programming, where needed, with the safety of statically declared and enforced types, where possible. We integrate ideas for statically typing strategies by Lämmel (2003) with ideas from the gradual typing literature (Siek and Taha 2006; Siek and Taha 2007). In particular, we extend conventional static types with the special type for type preserving transformations (Lämmel 2003). And we introduce a dynamic type in the tradition of gradual type systems to account for, as yet, untyped code. At the interface of statically and dynamically typed code, the type checker inserts dynamic type checks (through casts and proxies) to guarantee the assumptions of static code. This ensures that the type system is backwards compatible such that existing code can pass the type checker as is, and such that types can be introduced incrementally to existing code. At the intersection of typed strategies and gradual types, we find an interesting dynamic types for strategies. For example, the type unifying strategies of Lämmel (2003) do not need a special type, but can be modelled with a dynamic input type and a specific result type.

The four contributions of this chapter are:

- We motivate and validate the gradual type system for Stratego with idiomatic examples (Sections 4.2 and 4.3).
- We formally define a type system for Core Gradual Stratego, which combines

static types for generic traversals with gradual types for partially typed programs (Section 4.4). The combination supports the (partially) dynamic typing of strategic programming patterns that are inherently not (completely) statically typable. This constitutes the first static type system for the Stratego language.

- We have implemented a type checker based on the type system, which reports violations in the IDE and which generates code with casts and proxies for type checking at run-time at the interface of statically and dynamically checked code (Section 4.4).
- We evaluate the application of the type checker on an existing Stratego codebase to which we added type annotations (Section 4.5).

4.2 Rewriting Strategies and Types

Stratego is a language for the definition of program transformations with rewrite rules and programmable rewriting strategies (E. Visser, Benaissa and Tolmach 1998). In this section we give a (non-exhaustive) introduction to Stratego by means of examples as motivation for a type system. We give two encodings of the same transformation. The first encoding is completely statically typable. The second encoding requires dynamic typing. Furthermore, we analyse typical type errors.

4.2.1 *Program Transformation with Stratego*

The main ingredients of Stratego programs are algebraic signatures, rewrite rules, and strategies.

Algebraic Signatures. A Stratego program defines transformations on abstract syntax trees represented using first-order terms. The structure of terms is defined by means of an *algebraic signature*, which introduces sorts (types) and constructors on those sorts. A constructor declaration $c : t_1 * ... * t_n \to t_0$, defines a constructor c with the sorts of its arguments and result. A constructor declaration $: t_1 \to t_0$ is an *injection* of t_1 into t_0. This means a value of t_1 can be used directly as a value of t_0 without a constructor. Listing 4.2 defines the signature of a small imperative language with expressions and statements. An example of a well-formed term of sort `Stat` would be:

```
Seq(Assign("x", Add(Var("x"), Int("1"))), Lt(Var("x"), Int("3")))
```

Note that sorts and constructors are separate namespaces. In Spoofax, signatures are generated from a syntax definition in SDF3 (de Souza Amorim and E. Visser 2020), directly describing the structure of abstract syntax trees produced by parsers.

Rewrite Rules. Basic transformations are defined using *term rewrite rules*. A rewrite rule has the form $l : t_1 \to t_2$ with label l, left-hand side term t_1 and right-hand side term t_2. Applying a rewrite rule with label l to a term t means matching the term against the left-hand side term t_1, binding the variables in that term to sub-terms of t, and then replacing term t with the instantiation

```
rules
  desugar : // Exp to Exp
    Min(e) -> Sub(Int("0"), e)

  desugar : // Stat to Stat
    For(x, e1, e2, s) ->
    Seq(Assign(x, e1),
        While(Lt(x, e2),
              Seq(s, Assign(x, Add(x, Int("1"))))))

  desugar : // Exp to Exp
    Inc(x) -> Stat(Assign(x, Add(x, Int("1"))), x)

  desugar : // Stat to Stat; lift Stat from Exp
    stat@<is-simple-stat> -> Seq(s1, s2)
    where <oncetd-hd((Stat(s1, e) -> e))> stat => s2
```

Listing 4.1 Rewrite rules

```
signature
  sorts Var Exp constructors
    Var  : string -> Var
         : Var -> Exp
    Int  : string -> Exp
    Add  : Exp * Exp -> Exp
    Sub  : Exp * Exp -> Exp
    Lt   : Exp * Exp -> Exp
    Min  : Exp -> Exp
    Inc  : Var -> Exp
    Stat : Stat * Exp -> Exp
  sorts Stat constructors
    Exp    : Exp -> Stat
    Skip   : Stat
    Assign : Var * Exp -> Stat
    Seq    : Stat * Stat -> Stat
    While  : Exp * Stat -> Stat
    For    : Var * Exp * Exp * Stat -> Stat
```

Listing 4.2 Signature

of the right-hand side t_2. If the match fails, the rule fails to apply. If the specification has multiple rules with the same name, they are tried in order.

A rule `l : t_1 -> t_2 where s` is a conditional rule, where `s` is a strategy expression. When applying a conditional rule, the condition `s` is applied to the subject term, using variables bound in the match of the left-hand side, and possibly binding variables that are used in the right-hand side. When the condition fails, applying the rule fails.

Listing 4.1 shows examples of rewrite rules for a desugaring transformation on expressions and statements of the language of Listing 4.2. The first two rules define `Min` and `For` in terms of other constructs. The third rule defines increment (think C-style post increment `x++`) in terms of assignment to the variable incremented. Since `Inc(_)` is an expression and `Assign` is a statement, replacing one by the other would not be well-formed. The `Stat` constructor defines an expression form that allows embedding a statement within an expression.

The fourth rule defines lifting of statements embedded in expressions to the statement level. The match pattern uses a guard that checks that the term is a simple statement (defined in Listing 4.3), and the condition of the rule applies a left-most depth-first ('once-top-down') traversal that finds an embedded occurrence of a term `Stat(s1, e)`, replacing that occurrence with the expression, and binding the statement to variable `s1`. The syntax for the match is `n @ t` to bind variable `n` to the term `t` and `< s >` to apply strategy `s` as a guard to the term, i.e. it only matches if the strategy succeeds on the given term. The argument of `oncetd` is an *anonymous* rewrite rule `(Stat(s1, e) -> e)` that does not scope its variables. Thus, bindings are available in the context of the traversal. (Such contextual binders were introduced by E. Visser, Benaissa and Tolmach (1998) and later generalised to *dynamic rewrite rules* by (Bravenboer, van Dam et al. 2006).) For example, the rules give rise to a sequence of transformations such as the following:

```
    Assign(Var("x"), Min(Inc(Var("x"))))
--> Assign(Var("x"), Min(Stat(Assign(Var("x"),
                         Add(Var("x"), Int("1"))),
                         Var("x"))))
--> Assign(Var("x"), Sub(Int("0"),
                    Stat(Assign(Var("x"),
                         Add(Var("x"), Int("1"))),
                         Var("x"))))
--> Seq(Assign(Var("x"), Add(Var("x"), Int("1"))),
        Assign(Var("x"), Sub(Int("0"), Var("x"))))
```

Rules are not applied automatically, but their application (order) is determined by a strategy.

Strategies. Rewrite rules transform a term into another term. Traditional rewrite systems apply such rules exhaustively throughout a term. Stratego provides *programmable strategies* for ordering the application of rewrite rules to a term. For example, Listing 4.3 defines the `transform` strategy to apply the `desugar` rules of Listing 4.1 using a bottom-up strategy that tries to apply the rules at each node. Strategies such as `bottomup` and `try` are not built-in, but generic, parametric strategies defined in terms of basic *strategy combinators*.

Stratego's built-in strategy combinators include identity `id`, failure `fail`, sequential composition `s1; s2`, and ordered choice `s1 <+ s2`. Matching (and returning) a term pattern (`?t`) and instantiating (aka building) a term pattern (`!t`) are first-class citizens. The expression `<s>t` applies strategy `s` to term `t` and the expression `s => t` matches the result of `s` against term `t`. The generic traversal combinators `all(s)` and `one(s)` apply a transformation to all, respectively, one, of the sub-terms of a term. Given a constructor $c : t_1 * ... * t_n \rightarrow t_0$, a corresponding *congruence traversal* strategy $c(s_1, ..., s_n)$ transforms c-terms, applying the corresponding strategies to the sub-terms.

Listings 4.3 and 4.4 use these combinators to define strategies[1]. Strategy `is-simple-stat` determines whether a term is a simple statement through pattern matching. The traversal strategy `oncetd-hd` uses congruence traversal to

[1]Note that there is no strict syntactic separation between rules and strategies. The section headers are used to indicate intention.

```
strategies
  transform = bottomup(try(desugar))

  is-simple-stat = ?Assign(_,_) <+ ?Exp(_)
    <+ ?While(_,_) <+ ?For(_,_,_,_)

  oncetd-hd(s) =
    While(oncetd(s),id) <+ For(id,oncetd(s),id,id)
    <+ For(id,id,oncetd(s),id) <+ Exp(oncetd(s))
    <+ Assign(id, oncetd(s))
```

Listing 4.3 Strategies

```
strategies
  try(s)      = s <+ id
  topdown(s)  = s; all(topdown(s))
  bottomup(s) = all(bottomup(s)); s
  oncetd(s)   = s <+ one(oncetd(s))
  alltd(s)    = s <+ all(alltd(s))
```

Listing 4.4 Generic strategies

apply a `oncetd(s)` traversal to selected arguments of constructors (the 'heads'). Listing 4.4 defines several generic strategies. The strategy `try(s)` applies `s` and when that fails succeeds with the original term. The strategy `bottomup(s)` first visits the direct sub-terms of the subject term with a recursive call and then applies `s` to the result. Strategy `oncetd(s)` transforms the first term for which `s` succeeds in left-most depth-first traversal. Strategy `alltd(s)` applies `s` to all outermost terms for which `s` succeeds.

Non-Well-Formed IR. The rules defined in Listing 4.1 take care to only replace terms with terms of the same sort. The `Stat` constructor is used to embed a statement within an expression. While this is good practice, it is not required. Listing 4.5 shows an alternative approach to the transformation. The transformation is defined in two stages. In the first stage, the `desugar-inc` rule replaces `Inc` terms with assignments, creating non-well-formed intermediate terms. In the second stage, the `lift-assign` rule lifts assignments embedded in expressions to assignment level. While intermediate terms are not well-formed with respect to the signature, after applying `transform`, terms are well-formed again.

4.2.2 Type Errors

Stratego is a memory-safe, dynamically typed language. The Stratego runtime, in collaboration with the code generator or interpreter, ensures that a program that passes the static checks, does not crash. A program may terminate with a transformed term, with a (pattern match) failure, or with an exception. (An alternative version of conditional rules requires that the condition succeeds and raises an exception if it does not.)

The front-end of the compiler applies some static checks. Constructors need to be declared and used with the arity corresponding to the declaration. Rules and strategies need to be defined when called. Variables should be bound

```
rules
  desugar-inc : // Exp to Stat
    Inc(x) -> Assign(x, Add(x, Int("1")))

  lift-assign : // lift Stat from Exp
    s1@<is-simple-stat> -> Seq(Assign(x, e), s2)
    where <oncetd-hd((Assign(x, e) -> x))> s1 => s2

strategies
  transform   = desugar-all; lift-all
  desugar-all = bottomup(try(desugar-inc))
  lift-all    = alltd(lift-assign; lift-all)
```

☙ **Listing 4.5** Alternative rules and strategies

```
rules // errors caught by current compiler
  desugar :
    Inc(Vaz(x)) -> // unknown constructor
    Stat(Assign(y, // unbound variable
                Add(Var(x), Int("1"))))
  // constructor Stat used with wrong arity
  desugar-some =
    top-down(desugar) // unknown strategy

rules // errors not caught by current compiler
  desugar : // expression replaced with statement
    Stat(stat, e) -> stat

  desugar : // lifting also applied to expressions
    stat -> Seq(s1, s2)
    where <oncetd((Stat(s1, e) -> e))> s => s2

  desugar :
    Inc(x) -> Stat(Assign(Var(x),         // Var instead of string
                          Add(x,Int(1))), // int instead of string
                          x)
```

☙ **Listing 4.6** Rules and strategies with errors

when used in a build pattern. Otherwise, the compiler is fairly permissive, allowing programs such as in Listing 4.5. In Listing 4.6 we give examples of errors that are caught by the Stratego compiler and errors that are not caught by the compiler nor by the runtime.

A typical consequence of the lack of static typing is that a transformation is applied successfully, but constructs a non-well-formed term. Subsequently, a pretty-printer (e.g., generated from a syntax definition (de Souza Amorim and E. Visser 2020)) transforming terms to Box expressions, fails because the term does not match the expected abstract syntax schema. This leads to expensive debugging sessions to track down the origin of the non-wellformedness. A type system for Stratego that identifies such errors statically will make Stratego programming much more productive at micro scale. At macro scale, having types for interfaces will make code more maintainable. Furthermore, the compiler could benefit from the guarantees of static types in optimisations. At the same time, such a type system should not prevent the usage of existing code.

4.3 Gradually Typing Strategies

We introduce a type system for Stratego that addresses the lack of static type checking discussed in the previous section. In this section, we first discuss the requirements for a type system, and then we give a high-level overview of the type system building on the examples of the previous section. In the next section, we formalise the type system.

4.3.1 Requirements

The design and implementation of a type system for Stratego should satisfy the following requirements. It should be backward compatible such that existing programs are accepted as is, with the same run time semantics. It should impose minimal type annotation requirements on programs to preserve the concise style of Stratego programs. It should support generic traversal primitives, providing static typing where possible, but also support dynamic usage. It should support a simple migration path from untyped to typed code. It should be modularly checkable for integration into the incremental compiler of Chapter 3. It should have limited negative impact on performance. In the rest of this chapter, we describe a type system that mostly meets these criteria; we have not evaluated performance yet.

4.3.2 Types for Stratego

We first discuss complete static checking. We will write strategy to indicate both rules and strategies.

Top-Level Type Annotations. To force checking the type of a strategy, one explicitly declares its type using a type annotation. For example, we can declare the types of `transform` and `desugar-inc` from the previous section as:

```
transform :: Stat -> Stat
desugar-inc :: Exp -> Stat
```

The caller of a strategy with a type annotation needs to ensure that the term that it is applied to has the right type. The type checker checks that given the assumption on the input term, the strategy definition guarantees that the output term has the specified type, or that the strategy fails. The type system does not infer types for top-level declarations of strategies. If a type is omitted type dynamic is assumed, as we will discuss below.

Note that the addition of a top-level type can invalidate code that is acceptable when dynamically typed.

Type Checking Term Patterns. Given an expected input or output type, the type checker checks that a pattern is well-formed with respect to that type and the constructors in the signature. Listing 4.7 on page 98 shows examples of errors caught in pattern matching and instantiation. These would not be errors if the type annotation was not present to restrict the strategies to a specific type. Thus, the errors in Listing 4.6 in the desugaring rule for `Inc` are all caught.

Inferring Types of Variables. While we opted to not infer types for top-level declarations, we do infer types for local variables. Variables in Stratego rules are declared implicitly by using them in a match position. To avoid clutter, we do not require explicit declaration of type annotations for local variables, but rather infer their type from context, where we take the types of the left- and right-hand sides of a rule as leading. For example, consider the `ssa` rules in Listing 4.8. The types of the local variables in the condition of the rule follow from the type of the strategy, and then from the types inferred in previous steps in the condition. Consider the errors that are found when the variables in the `Stat(_,_)` pattern match are swapped:

```
ssa : Min(e) -> s2
  where <ssa> e => Stat(e', s1)
  ; !Var(<new>) => x
  ; !Stat(Seq(s1,                // warn: not Stat
              Assign(x, Min(e'))),// warn: not Exp
          x) => s2               // error: not Stat
```

Type Preserving Transformations. Generic term traversals, such as `bottomup`, apply an arbitrary transformation to the sub-terms of a term. As we illustrated in Listing 4.5, such generic traversals may construct (temporarily) non-well-formed terms. However, a particular class of traversals is more well-behaved and transforms each term to a term of the same type (or fails), possibly operating heterogeneously on multiple types. In other words, such strategies are *type preserving*. Following the work of Lämmel (2003) we introduce the `TP` type to characterise type preserving strategies.

Type preserving transformations can be heterogeneous rewrite rules such as `desugar` in Listing 4.1, which operate on terms of different types, but ensure to always transform each term to a term of the same type[2]. For example, `desugar` transforms `Exp` terms to `Exp` terms, and `Stat` terms to `Stat` terms. Generic strategies can construct type preserving strategies from type preserving strategies. For example, if `s` is of type `TP`, then `try(s)` is also of type `TP`. Similarly, `all(s)` is `TP` if `s` is, and `s1; s2` is `TP` if `s1` and `s2` are. Given these ingredients and the annotation `bottomup(TP) :: TP`, we can conclude that the definition `bottomup(s) = all(bottomup(s)); s` is well-typed, and that `bottomup` is a type preserving strategy.

Given the type annotations in Listing 4.9 on page 99, the rules in Listing 4.1 and the strategies in Listings 4.3 and 4.4 are completely statically typable, except for one detail, which we discuss next.

Type Match. The rule `min : e -> Min(e)` takes any term and applies `Min` to it. That is, its applicability is not guarded by a pattern with a constructor. For example, in untyped Stratego we can write `<min>Skip() => Min(Skip())`, producing a non-well-formed term. When we give it the type annotation `min :: Exp -> Exp` we express that *when applied to an `Exp` it will return an `Exp`*. The caller should guarantee to only apply it to `Exp` terms. In order to qualify for a type annotation `min :: TP`, stronger requirements apply. A `TP` strategy should

[2]The difference between `TP` and polymorphic strategy `a -> a` is discussed at the end of Section 4.4.

```
strategies
  is-stat :: Stat -> Stat
  is-stat = ?Assign(_,_)        // ok
  is-stat = ?Var(_)             // error: not a Stat
  is-stat = ?Seq(Var(_),_)      // error: arg not Stat
  mk-stat :: () -> Stat
  mk-stat = !Exp(Var("x"))      // ok
  mk-stat = !Var("x")           // error: not a Stat
  mk-stat = !Exp(Exp(Var("x")))// error: arg not Exp
```

Listing 4.7 Checking (nested) patterns

```
rules
  new :: () -> string
  ssa :: Exp -> Exp
  ssa : Var(x) -> Stat(Skip(), Var(x))
  ssa : Min(e) -> s2
    where <ssa> e => Stat(s1, e')
    ; !Var(<new>) => x
    ; !Stat(Seq(s1, Assign(x, Min(e'))), x) => s2
```

Listing 4.8 Inferring types of variables

be applicable to a term of any type and produce a well-formed term of the same type as output (or fail). Thus, a `TP` strategy should check its own input type requirements. Given that the new type checker relies on dynamic type checks for casts (see below), we also make this functionality available as a *type match*. For any declared sort t, the strategy `is(t)` *dynamically* checks that the subject term is of type t and *statically* guarantees that that is the case when it succeeds. Thus, we can define the `min` rule as `min : e@<is(Exp)> -> Min(e)` and give it the `TP` annotation.

In Listing 4.3 we defined `is-simple-stat` to identify simple statements. Such strategies are often used to define fine grained recognisers of subsets of types (E. Visser 1999). Unfortunately, its use in the `desugar` rule in Listing 4.1 on page 92 is not sufficient to convince the type checker that the rule is `TP`. Using the type match `is(Stat)` we can convey that it is:

```
  desugar :
    s@<is(Stat); is-simple-stat> -> Seq(s1, s2)
    where <oncetd-hd((Stat(s1, e) -> e))> s => s2
```

With that edit, the rules in Listing 4.1 and the strategies in Listings 4.3 and 4.4 are completely statically typable against the type annotations in Listing 4.9.

Polymorphic Strategies. Our type checker supports parametric polymorphic types for rules and strategies. We use prenex polymorphism with lowercase names as type variables, which was already used informally (i.e. without type checker support) in signatures. In Listing 4.10 on page 100 we define several polymorphic strategies on lists from the standard library (edited for space) and provide type annotations for them. Note that type match and imperative update make that we can not support parametricity (Reynolds 1983; Wadler 1989).

```
rules
  transform :: Stat -> Stat
  desugar :: TP
  try(TP) :: TP
  bottomup(TP) :: TP
  oncetd(TP) :: TP
  alltd(TP) :: TP
```

Listing 4.9 Type annotations

4.3.3 Gradual *Types for Stratego*

Not all Stratego programs can be statically type checked using the techniques discussed above. For example, the alternative transformation approach of Listing 4.5 constructs intermediate terms that are not well-formed with respect to the signature. Furthermore, there exists a significant amount of Stratego code without type annotations. Imposing the requirement that all strategies should be annotated, before the new type checker can be used would be prohibitive. This is one of the classical motivations for the introduction of gradual types (Siek and Taha 2006).

We have extended the type system sketched above with dynamic types such that *any* existing Stratego program (that passes the static checks of the legacy compiler), will pass the type checker and will have the same runtime semantics. (Note that Stratego's syntax already enforces a distinction between strategies (functions) and terms.)

Type Dynamic and Type Casts. We extend the set of types with the type dynamic ?, which represents a dynamically checked type (not to be confused with the match operator). When a typed strategy is invoked on a dynamically typed term, the term needs to be checked in order to guarantee the input requirements. This check is done by a type cast, an assertion that verifies that the subject term has the expected type. Failing the assertion is a programming error and leads to an exception. The type checker (silently) inserts casts where needed, in the style of gradual typing (Siek and Taha 2006). When a top-level strategy does not have a type declaration, it is considered to have a dynamic type. The dynamic type can also be used explicitly in type annotations.

Gradually Typing Term Patterns. Term patterns in Stratego appear in either matching or building position. When we match against a dynamically typed term, we can not assume that it is a well-formed term. Therefore, we only check that the constructors that are used in the pattern are defined and have the right arity, compatible with the legacy Stratego compiler. When we build a term that is expected to be dynamically typed, we also check that the used constructors are defined and have the right arity. However, we can sometimes infer the type of a well-formed term in a build pattern (Section 4.4). This extra type information is propagated, and can prevent some unnecessary casts from being inserted.

Proxies. The type checker must also type check strategy arguments to other strategies. We do not support higher-order casts, i.e. type assertions on

```
rules
  filter(a -> b) :: List(a) -> List(b)
  filter(s) : [] -> []
  filter(s) : [x | xs] -> <conc> (<opt(s)> x, <filter(s)> xs)

  opt(a -> b) :: a -> List(b)
  opt(s) = ![<s>] <+ ![]

  conc :: List(a) * List(a) -> List(a)
  conc : ([], xs) -> xs
  conc : ([x | xs], ys) -> [x | <conc> (xs, ys)]

  mapconc(a -> List(b)) :: List(a) -> List(b)
  mapconc(s) : [] -> []
  mapconc(s) : [x|xs] -> <conc> (<s> x, <mapconc(s)> xs)
```

Listing 4.10 Polymorphic strategies

```
strategies
  if-odd(Nat -> Nat, Nat -> Nat) :: Nat -> Nat
  if-odd(s1, s2): Z() -> <s2>
  if-odd(s1, s2): S(t) -> <if-even(s2, s1)> t

  if-even(s1, s2): Z() -> <s1>
  if-even(s1, s2): S(t) ->
    <if-odd(cast(Nat); s2; cast(Nat),
            cast(Nat); s1; cast(Nat))> t
  // proxy version:
  if-even(s1, s2): S(t) ->
    <if-odd(proxy(|Nat,Nat|s2),
            proxy(|Nat,Nat|s1))> t
```

Listing 4.11 Explicit casts surrounding strategy arguments can accumulate into closures inside closures, a space and time leak we should avoid.

strategies, directly. Instead we do this dynamic type check lazily, at the time the strategy is called. We do so by creating a new closure for a strategy argument that includes a type cast. But if a strategy argument is passed through a couple of statically and dynamically typed strategies, this can lead to an accumulation of closures in closures, as noted by Herman, Tomb and Flanagan (2010). Listing 4.11 shows an example. When `<intToNat; if-even(!Z(), !S(Z()))> 10` is executed, the first argument is wrapped in a type cast closure *five* times before it is executed. Therefore, the type checker inserts type proxies instead of normal casts. These are closures that contain a type assertion for the input and output of a strategy, and the strategy (closure) itself, as a special value that the runtime can inspect. When a new proxy is constructed around a proxy value, these are collapsed into one proxy value to avoid the accumulation of closures in closures:

```
proxy(|Nat,Nat|proxy(|Nat,Nat|s1))
--> proxy(|Nat;Nat,Nat;Nat|s1)
--> proxy(|Nat,Nat|s1)
```

Locally required type casts can be merged with the type casts of the proxy value. This makes dynamic type checks less lazy, as incompatible type assertions can

```
rules
  collect(? -> b) :: ? -> List(b)
  collect(s) = // all sub-terms for which s succeeds
    <conc>(<opt(s)>, <kids; mapconc(collect(s))>)
  kids :: ? -> List(?) // list of children, using
  kids : _#(xs) -> xs // generic term deconstruction

strategies
  vars :: ? -> List(Var)  // all variables
  vars = collect(?Var(_)) // in a program
```

Listing 4.12 Type unifying strategies

be found while adding them to an existing proxy, similar to the work of Siek, Garcia and Taha (2009).

Type Preserving and Dynamically Typed Traversals. While the `TP` type allows static typing of many transformations, the type restricts the legitimate use of useful standard library strategies. To prevent duplication of type preserving and dynamically typed versions of the same strategies, the type checker allows a dynamically typed fall back type annotation for a strategy. For example, the `bottomup` strategy can also be typed with the more permissive type annotation `bottomup(? -> ?) :: ? -> ?`. This strategy can be called on any term, well-formed or not, since the strategy argument is given no guarantees on what kinds of terms it is called upon. Perhaps a more interesting case is the `try` strategy, which still has some typing even when not type preserving: `try(a -> b) :: a -> ?`. A strategy wrapped in a try must still be called with the right input type for the strategy.

Type Unifying Strategies. Lämmel (2003) coined the term *type unifying strategies* (and the special type `TU(b)`) for Stratego strategies that take any input and return a single, typed output. We use `?` for the input to allow any input and a type parameter for the output. Listing 4.12 shows an example of a type unifying generic traversal.

Polymorphism Revisited. The prenex polymorphism in our type system has a limitation. The current runtime of Stratego does not support strategies with explicit type arguments. Therefore, type arguments must be completely abstract, and cannot be used in dynamic type assertions. Our type system gives a type error when a cast with a type variable must be inserted. This means that polymorphic strategies and rules are less gradual than the rest of gradually typed Stratego.

4.4 A Type System for Core Gradual Stratego

In this section, we present an algorithmic type system for Core Gradual Stratego, which formalises the ideas of the previous section. Previous work on formalisations of Stratego were based on System S, the calculus of strategy combinators (E. Visser and Benaissa 1998; Lämmel 2003). We consider a larger core language, including signatures and definitions of rules and strategies, as these matter for the type system.

$$
\begin{array}{rcll}
sl & ::= & \textit{string literals} & \\
f,x & ::= & \textit{names} & \\
o & ::= & f\ \texttt{:}\ ot \ \mid\ \texttt{:}\ ot & \text{signatures} \\
ot & ::= & t \ \mid\ \bar{t}\ \rightarrow\ t & \text{sig types} \\
t & ::= & f\texttt{(}\bar{t}\texttt{)} \ \mid\ \texttt{string} \ \mid\ \texttt{?} \ \mid\ \texttt{ill-formed} & \text{types} \\
st & ::= & \texttt{(}\overline{st}\texttt{)}\ t\ \rightarrow\ t \ \mid\ \texttt{?} & \text{strategy types} \\
d & ::= & f\texttt{(}\bar{f}\texttt{)}\ \texttt{=}\ s & \text{definitions} \\
 & \mid & f\texttt{(}\bar{f}\texttt{)}\ \texttt{:}\ e \rightarrow e\ \texttt{where}\ s & \\
s & ::= & f\texttt{(}\bar{s}\texttt{)} \ \mid\ \texttt{fail} \ \mid\ \texttt{id} \ \mid\ \texttt{?}e \ \mid\ \texttt{!}e & \text{strategies} \\
 & \mid & \texttt{\{}\ x\ \texttt{:}\ s\ \texttt{\}} \ \mid\ s\ \texttt{;}\ s \ \mid\ s\ \texttt{<}\ s\ \texttt{+}\ s & \\
 & \mid & \texttt{cast(}c\texttt{)} \ \mid\ \texttt{proxy(}\overline{sc}\texttt{|}c\texttt{,}c\texttt{|}s\texttt{)} & \\
e & ::= & x \ \mid\ \texttt{\_} \ \mid\ sl \ \mid\ f\texttt{(}\bar{e}\texttt{)} \ \mid\ e\ ::\ t & \text{terms} \\
c & ::= & \texttt{id} \ \mid\ t & \text{coercions} \\
sc & ::= & \texttt{id} \ \mid\ st & \text{strategy coercions}
\end{array}
$$

Figure 4.1 The Stratego core grammar. Any Stratego program can be desugared to these core constructs.

4.4.1 Core Gradual Stratego

Figure 4.1 defines the grammar of Core Gradual Stratego, which is Core Stratego extended with (dynamic) types, top-level type annotations, casts, and proxies. We use vector notation instead of a Kleene star for lists, to mirror the type rules. These are all zero-or-more, except for the second alternative of *ot*, which is one-or-more.

Signatures *o, ot* consist of constructor definitions with zero or more arguments, and injection definitions (an example is back in Listing 4.2 on page 92). Types *t*, or sorts, include built-in types (string, int) and (parameterised) types. We extend types with the dynamic type ?, and the ill-formed type. We also add strategy types *st* to describe strategy arguments.

Definitions *d* include strategy definitions and rule definitions. Including rewrite rule in the core is not necessary for dynamic expressivity; usually they are presented as sugar, by translation to a strategy sequence of a match, side-condition, and a build (E. Visser and Benaissa 1998). However, we give type annotated rules a more intuitive typing.

Strategy expressions *s* consist of calls to strategies or rules, the explicit match failure strategy, the identity strategy, match, build, scoping a variable, sequences, guarded choice, and finally the additions, cast and proxy (note that the two vertical bars that are part of the syntax of proxy). Term expressions *e* consist of variables, wildcards, string literals, constructor applications, and additionally type ascription. Casts and proxies apply coercions, which can be a coercion *c* to a type or the identity coercion.

4.4.2 Algorithmic Type System

We present an algorithmic type system, written as a transformation on the program that inserts casts while type checking. The type system follows an

Type coercion rules $\boxed{\Gamma \vdash t \rightsquigarrow t : c}$

$$\frac{\Gamma \vdash t_1 <: t_2}{\Gamma \vdash t_1 \rightsquigarrow t_2 : \texttt{id}}\ [\textsf{csub}] \qquad \frac{}{\Gamma \vdash \texttt{?} \rightsquigarrow t : t}\ [\textsf{ccheck}]$$

Subtype rules $\boxed{\Gamma \vdash t <: t}$

$$\frac{}{\Gamma \vdash t_1 <: \texttt{?}}\ [\textsf{dyntop}] \qquad \frac{}{\Gamma, t_1 <: t_2 \vdash t_1 <: t_2}\ [\textsf{subinj}]$$

$$\frac{\Gamma \vdash \overline{t_1} <: \overline{t_2}}{\Gamma \vdash f(\overline{t_1}) <: f(\overline{t_2})}\ [\textsf{subcovar}] \qquad \frac{}{\Gamma \vdash t <: t}\ [\textsf{subrefl}]$$

Figure 4.2 Algorithmic coercion and subtype rules. Alternative rules are tried in order.

abstract interpretation approach.

Meta-properties. We currently do not have a formal dynamic semantics for Core Gradual Stratego, but assuming a reasonable dynamic semantics as sketched in previous sections, we postulate that the type system presented here is sound for the typed subset of the core language, i.e. all programs accepted by the type system execute without type-errors. Furthermore, if a program after cast insertion executes without type-errors, we postulate that the same program without type annotations (but preserving type tests) will also execute and give the same result. In other words, type annotations do not affect the behaviour or results of type-correct programs. Finally, we believe that the presented transformation rules are deterministic if alternatives are tried in order, and that their execution terminates for all inputs.

Environments. The type system is meant to be modular, therefore we assume that environment Γ contains all available definitions and their types, so we can resolve calls. The environment also contains the types of defined constructors and the precomputed transitive relation of defined injections, i.e. every mapping from one type into another without constructor becomes a subtype fact, to be available for the [subinj] rule of Figure 4.2. Most of the information in the environment flows through the type system like a store. Local variables can be bound to different types at different points in the program, e.g. inside and after a guarded choice. The arity of dynamically typed strategy arguments is discovered during type checking.

Coercion, Subtyping and Bounds. Figure 4.2 defines the computation of coercions from one type to another. The [csub] rule defers to subtyping. The coercion from a subtype to its supertype is the identity coercion. The [ccheck] rule defines that it is also possible to go from the ? type to any specific type by coercing to that type.

The subtyping rules in Figure 4.2 define that ? is a supertype of any other type ([dyntop]). The [subinj] rule looks up an injection in the environment. The [subcovar] rule defines that parameterised sorts are co-variant in their

Definition type rules $\boxed{\Gamma \vdash d \Rightarrow d \dashv \Gamma}$

$$\frac{\begin{array}{c} len(\overline{f}) = j \qquad \Gamma_1 = \Gamma_1', \langle f, j \rangle : (\overline{st})\ t_1 \to t_2 \\ \Gamma_1, \overline{f \mathrel{\bar{:}} st}; t_1 \vdash s_1 \Rightarrow s_2 \dashv \Gamma_2; t_3 \qquad \Gamma_2 \vdash t_3 \rightsquigarrow t_2 : c \end{array}}{\Gamma_1 \vdash f(\overline{f})\ \texttt{=}\ s_1 \Rightarrow f(\overline{f})\ \texttt{=}\ s_2\texttt{;}\ \texttt{cast}(c) \dashv \Gamma_2}\ [\text{sdef}]$$

$$\frac{\begin{array}{c} len(\overline{f}) = j \qquad \Gamma_1 = \Gamma_1', \langle f, j \rangle : (\overline{st})\ t_1 \to t_2 \\ \Gamma_1, \overline{f \mathrel{\bar{:}} st}; t_1 \vdash_m e_1 \Rightarrow e_3; s_2 \dashv \Gamma_2; t_3 \\ \Gamma_2; t_2 \vdash_b e_2 \Rightarrow e_4 \dashv \Gamma_3; t_2' \qquad \Gamma_3; t_3 \vdash s_1 \Rightarrow s_3 \dashv \Gamma_4; t_3' \end{array}}{\Gamma_1 \vdash f(\overline{f})\ :\ e_1 \to e_2\ \texttt{where}\ s_1 \Rightarrow f(\overline{f})\ :\ e_3 \to e_4\ \texttt{where}\ s_2\texttt{;}\ \ s_3 \dashv \Gamma_4}\ [\text{rdef}]$$

Figure 4.3 Algorithmic typing rules for adding definitions to the environment. Alternative rules are tried in order.

Strategy typing $\boxed{\Gamma; t \vdash s \Rightarrow s \dashv \Gamma; t}$

$$\frac{\Gamma_1; t_1 \vdash_m e_1 \Rightarrow e_2; s \dashv \Gamma_2; t_2}{\Gamma_1; t_1 \vdash \texttt{?}e_1 \Rightarrow \texttt{?}x\texttt{@}e_2\texttt{;}\ s\texttt{;}\ \texttt{!}x \dashv \Gamma_2; t_2}\ [\text{match}] \qquad \frac{\Gamma_1; t_1 \vdash_b e_1 \Rightarrow e_2 \dashv \Gamma_2; t_2}{\Gamma_1; t_1 \vdash \texttt{!}e_1 \Rightarrow \texttt{!}e_2 \dashv \Gamma_2; t_2}\ [\text{build}]$$

$$\frac{\begin{array}{c} len(\overline{s_1}) = j \qquad \Gamma_1 = \Gamma_1', \langle f, j \rangle : (\overline{st})\ t_1 \to t_2 \\ \Gamma_1; \overline{st}\ \overset{\longrightarrow}{\vdash_{sa}}\ \overline{s_1} \Rightarrow \overline{s_2} \dashv \Gamma_2 \qquad \Gamma_2 \vdash t_0 \rightsquigarrow t_1 : c \end{array}}{\Gamma_1; t_0 \vdash f(\overline{s_1}) \Rightarrow \texttt{cast}(c)\texttt{;}f(\overline{s_2}) \dashv \Gamma_2; t_2}\ [\text{call1}]$$

$$\frac{\begin{array}{c} len(\overline{s_1}) = j \qquad \Gamma_1 = \Gamma_2, \langle f, 0 \rangle : \texttt{?} \\ \Gamma_2, \langle f, j \rangle : (\overline{\texttt{?}})\ \texttt{?} \to \texttt{?}; \overline{\texttt{?}}\ \overset{\longrightarrow}{\vdash_{sa}}\ \overline{s_1} \Rightarrow \overline{s_2} \dashv \Gamma_3 \end{array}}{\Gamma_1; t_1 \vdash f(\overline{s_1}) \Rightarrow f(\overline{s_2}) \dashv \Gamma_3; \texttt{?}}\ [\text{call2}]$$

Figure 4.4 Excerpt of the algorithmic typing rules for the core Stratego strategy expressions. Alternative rules are tried in order.

parameters. Subtyping is reflexive ([subrefl]), but not transitive. Instead we took the transitive closure of the injections before the start of the program.

With this definition we can define the least-upper-bound and greatest-lower-bound on types. The representative type is used for any two types from an injection cycle. In all other cases it is the expected bound wherever injections form a lattice. Note that injections do not guarantee a lattice structure, so there may not be unique bounds. In those cases where we do not find a unique bound, we use the `?` type.

Definitions. Figure 4.3 defines the typing judgements for rule and strategy definitions. Rules for definitions look up their type in the environment, and register their arguments and type variables. For strategy definitions [sdef], we then check the body, and insert a cast after the body with a coercion from the result type to the output type of the definition. We slightly abuse the syntax of vectors here for pairing the strategy arguments with their types ($\overline{f} \mathrel{\bar{:}} \overline{st}$). These strategy names are paired with their arity and those together with their type are put into the environment. Dynamically typed strategy arguments are put into the environment with arity 0, which is relevant later in Figure 4.6 on page 107.

For rule definitions [rdef] we check their input and output term expressions against the type definition first, before we check the side-condition. This entails that we may check variables in the output term that are bound in the side-condition. The rules for terms in build position take this into account. Since rules and strategies can be overloaded in the number of strategy arguments, and these remain separate definitions at run-time, the types of these definitions are associated to a pair of name and number of arguments.

```
strategies // [sdef] example
  assert-Stat :: ? -> Stat
  assert-Stat = id
// => assert-Stat = id; cast(Stat)
```

Strategies. We present an excerpt of the rules for strategy expressions in Figure 4.4; the full set of rules is defined in Appendix B. The typing judgement is $\Gamma; t \vdash s \Rightarrow s \dashv \Gamma; t$. These are the input context, type of the current term, and a strategy expression, which are transformed to a strategy expression with inserted casts, an output context, and the type of the current term after the strategy expression.

The [match] rule defers to the type rules for term expressions. Term expressions have a set of rules for match position and for build position. During a match of a term expression, there can be parts that need to be tested with a cast. Such casts are collected as a strategy expression, which is executed after the match. We use syntactic sugar to bind the current term to fresh variable x, and restore the current term value after the casts. The [build] rule defers to the build position term expression type rules, where casts can be inserted inline with some syntactic sugar.

For the strategy call we have two alternatives for typing. The [call1] and [call2] rules are attempted in order, the first to succeed is used. The [call1] rule looks up the strategy in the environment based on its name and arity. Then it type-checks the strategy arguments, and finally it computes the coercion required for the current type to match the input type of the found strategy. A cast is inserted before the call with that coercion. Of course an implementation of these rules may leave out the cast if the computed coercion is the identity coercion. The [call2] rule attempts to find the strategy under the arity 0 with the type ? in the environment. This is how strategy arguments are registered in the environment, for an untyped strategy with strategy arguments. If found, the environment is updated to reflect the discovered arity of the strategy argument.

The rules for calls work on vectors of arguments, and therefore need a form of iteration over these vectors. We put an arrow over the turnstile of a rule to denote that we map that particular rule over the vector arguments, and thread the other parts (typically the environment). For example:

$$\Gamma_1; ?, \overline{?} \;\overrightarrow{\vdash_{sa}}\; s_1, \overline{s_2} \Rightarrow s_3, \overline{s_4} \dashv \Gamma_3 \equiv$$
$$\Gamma_1; ? \vdash_{sa} s_1 \Rightarrow s_3 \dashv \Gamma_2 \wedge \Gamma_2; \overline{?} \;\overrightarrow{\vdash_{sa}}\; \overline{s_2} \Rightarrow \overline{s_4} \dashv \Gamma_3$$

Build Terms. Terms in build position are checked against the expected result type of the term (Figure 4.5). The resulting type is the type of the term that was actually built, while the environment can contain new bindings for local variables discovered in the term.

The rule for building string literals [bstring] defines that we can only have string literal where the expected type is a subtype of strings. The only subtypes of the built-in string type are aliases of the string type. The rule for type ascription [bascr] checks that the ascribed type is a subtype of the expected type, and propagates it to the subterm.

The rules for variables are again based on whether the variable was already bound to a type in the environment. If so, in [bvar1] we compute a coercion from the type of the variable to the expected type. We update the binding of the variable and the resulting type to the greatest-lower-bound of the two types. This can be done since the coercion to a subtype is only allowed if the supertype is `?`. In that we have discovered that after this build the variable must be the more specific type or the cast failed that we insert. The inserted cast uses syntactic sugar from the full Stratego language to be able to apply a strategy during the build of a term and put the result of that application there. If the variable that is built is not in the environment, rule [bvar2] is in effect, which will record the discovered type information. This will typically occur when we type-check the output term of a rule before the side-condition of the rule binds the variable.

Constructors can be built in contexts where a dynamically or a statically typed term is expected. The [bconstr1a] and [bconstr1b] rules handle the first case. With a dynamically typed term expected we look up a constructor of the right arity[3], and in [bconstr1a] we handle the case where despite the dynamically typed context, the child terms build the correct types for the constructor. In that case we can return the static type of the constructor. When the child terms do not build the correct types for the constructor, [bconstr1b] returns the **`ill-formed`** type, which is only a subtype of `?`. The [bconstr2] rule is the statically typed term construction, where we find a constructor of the right (sub)type for the expected output and require the child terms to have the correct child types. These constructor rules show another judgement with an adapted turnstile. In this case, with a bar over the turnstile, we have nothing to thread, but we lift the judgement point-wise over the vectors.

```
strategies // [rdef,bconstr2,bvar1] example
  exp-to-stat :: ? -> Stat
  exp-to-Stat: maybe-exp -> Exp(maybe-exp)
// => exp-to-Stat:
//      maybe-exp -> Exp(<cast(Stat)> maybe-exp)
```

Match Terms. Terms in match position are checked against the type of the term they are matched against. The rules are mostly analogous to those of build terms, and therefore elided here. For completeness, they are available in Appendix B.

[3]We use slightly different ⟨⟨brackets⟩⟩ for this pair to visually distinguish it from the ⟨strategy⟩ pair.

Build term typing $\boxed{\Gamma;t \ \vdash_b e \Rightarrow e \dashv \Gamma;t}$

$$\frac{\Gamma_1 \ \vdash t_2 \rightsquigarrow t_1 \ : \ c \qquad t_3 = t_1 \sqcap t_2}{\Gamma_1, x : t_2; t_1 \ \vdash_b x \Rightarrow \texttt{<cast(}c\texttt{)>} \ x \dashv \Gamma_1, x : t_3; t_3} \ \text{[bvar1]}$$

$$\frac{}{\Gamma_1; t_1 \ \vdash_b x \Rightarrow x \dashv \Gamma_1, x : t_1; t_1} \ \text{[bvar2]} \qquad \frac{\Gamma_1 \ \vdash t_1 <: \texttt{string}}{\Gamma_1; t_1 \ \vdash_b sl \Rightarrow sl \dashv \Gamma_1; t_1} \ \text{[bstring]}$$

$$\frac{\Gamma_1 \ \vdash t_2 <: t_1 \qquad \Gamma_1; t_2 \ \vdash_b e_1 \Rightarrow e_2 \dashv \Gamma_2; t_2'}{\Gamma_1; t_1 \ \vdash_b e_1 \ :: \ t_2 \Rightarrow e_2 \dashv \Gamma_2; t_2} \ \text{[bascr]}$$

$$\frac{\begin{array}{c} len(\overline{e_1}) = j \qquad \Gamma_1 = \Gamma_1', \langle\!\langle f, j \rangle\!\rangle : \overline{t_1} \rightarrow t_1 \\ \Gamma_1; \texttt{?} \ \overrightarrow{\vdash_b} \ \overline{e_1} \Rightarrow \overline{e_2} \dashv \Gamma_2; \overline{t_2} \qquad \Gamma_2 \ \overline{\vdash} \ \overline{t_2} <: \overline{t_1} \end{array}}{\Gamma_1; \texttt{?} \ \vdash_b f\texttt{(}\overline{e_1}\texttt{)} \Rightarrow f\texttt{(}\overline{e_2}\texttt{)} \dashv \Gamma_2; t_1} \ \text{[bconstr1a]}$$

$$\frac{len(\overline{e_1}) = j \qquad \Gamma_1 = \Gamma_1', \langle\!\langle f, j \rangle\!\rangle : \overline{t_1} \rightarrow t_1 \qquad \Gamma_1; \texttt{?} \ \overrightarrow{\vdash_b} \ \overline{e_1} \Rightarrow \overline{e_2} \dashv \Gamma_2; \overline{t_2}}{\Gamma_1; \texttt{?} \ \vdash_b f\texttt{(}\overline{e_1}\texttt{)} \Rightarrow f\texttt{(}\overline{e_2}\texttt{)} \dashv \Gamma_2; \texttt{ill-formed}} \ \text{[bconstr1b]}$$

$$\frac{\begin{array}{c} len(\overline{e_1}) = j \qquad \Gamma_1 = \Gamma_1', \langle\!\langle f, j \rangle\!\rangle : \overline{t_1} \rightarrow t_1 \\ \Gamma_1; \texttt{?} \ \overrightarrow{\vdash_b} \ \overline{e_1} \Rightarrow \overline{e_2} \dashv \Gamma_2; \overline{t_2} \qquad \Gamma_2 \ \vdash t_1 <: t_0 \qquad \Gamma_2 \ \overline{\vdash} \ \overline{t_2} <: \overline{t_1} \end{array}}{\Gamma_1; t_0 \ \vdash_b f\texttt{(}\overline{e_1}\texttt{)} \Rightarrow f\texttt{(}\overline{e_2}\texttt{)} \dashv \Gamma_2; t_1} \ \text{[bconstr2]}$$

Figure 4.5 Algorithmic typing rules for the core Stratego terms in build position. Alternative rules are tried in order.

Strategy argument typing $\boxed{\Gamma;st \ \vdash_{sa} s \Rightarrow s \dashv \Gamma}$

$$\frac{\Gamma_1 = \Gamma_1', \langle f, 0 \rangle : \texttt{?} \qquad \Gamma_1 \ \vdash \texttt{?} \rightsquigarrow t_2 \ : \ c}{\Gamma_1; \texttt{(}\overline{st_1}\texttt{)} \ t_1 \rightarrow t_2 \ \vdash_{sa} f\texttt{()} \Rightarrow \texttt{proxy(}\overline{\texttt{id}}\texttt{|id,}c\texttt{|}f\texttt{())} \dashv \Gamma_1} \ \text{[saref1]}$$

$$\frac{\begin{array}{c} \Gamma_1 = \Gamma_1', \langle f, 0 \rangle : \texttt{(}\overline{st_2}\texttt{)} \ t_3 \rightarrow t_4 \qquad \Gamma_1 \ \overline{\vdash} \ \overline{st_1} \rightsquigarrow \overline{st_2} \ : \ \overline{sc_1} \\ \Gamma_1 \ \vdash t_1 \rightsquigarrow t_3 \ : \ c_1 \qquad \Gamma_1 \ \vdash t_4 \rightsquigarrow t_2 \ : \ c_2 \end{array}}{\Gamma_1; \texttt{(}\overline{st_1}\texttt{)} \ t_1 \rightarrow t_2 \ \vdash_{sa} f\texttt{()} \Rightarrow \texttt{proxy(}\overline{sc_1}\texttt{|}c_1\texttt{,}c_2\texttt{|}f\texttt{())} \dashv \Gamma_1} \ \text{[saref2]}$$

$$\frac{\Gamma_1; t_1 \ \vdash s_1 \Rightarrow s_2 \dashv \Gamma_2; t_3 \qquad \Gamma_2 \ \vdash t_3 \rightsquigarrow t_2 \ : \ c}{\Gamma_1; \texttt{()} \ t_1 \rightarrow t_2 \ \vdash_{sa} s_1 \Rightarrow \texttt{proxy(|id,}c\texttt{|}s_2\texttt{)} \dashv \Gamma_2} \ \text{[sa]}$$

Figure 4.6 Algorithmic typing rules for strategy arguments in a call. Alternative rules are tried in order.

Strategy Arguments. Strategy arguments require special care (Figure 4.6). These arguments become closures, and if we add casts before we close over them, these casts will become invisible to the strategy that receives the arguments. A strategy argument that is passed around a few times can accumulate a number of casts, creating new closures for the cast and call sequence every time. This problem was identified and solved by Herman, Tomb and Flanagan (2010), by constructing special values containing the casts and the passed function. The values are called proxies, and they can be introspected at run-time. When creating a proxy directly around another proxy, no new proxy needs to be allocated. Instead the coercions are merged into the existing proxy. This saves memory, and also means that some of the dynamic type checks happen at that time, instead of when the proxy is executed.

In this core language, strategies have two kinds of arguments, the default term argument, and strategy arguments. Proxies save the coercions for the extra arguments, a coercion for input and output, and the strategy argument itself. The [saref1] rule handles strategy arguments that are a reference to dynamically typed strategies, [saref2] handles references to statically typed strategies (n.b. contravariance on the return type coercion), and [sa] handles other strategy expressions. Arbitrary strategy expressions need to become closures regardless of proxy objects. We can require the strategy expression to handle the exact type that it will receive as a strategy argument, but we still save the coercion on the return value in a proxy, so it is accessible at run-time.

Refer back to Listing 4.11 on page 100 for an example of proxy insertion.

4.4.3 *Polymorphism, Parametricity, and TP*

We have purposefully not discussed support for polymorphism in the type system so far. The language features of this core subset of Stratego have enough moving parts already. In this subsection we list the modifications to the type system that handle polymorphism. The environment now additionally holds pairs of type variables α and their type. Top-level definitions now have type schemes so the type variables can be made fresh at lookup time. Definitions register strategy arguments as definitions *without* type schemes. The subtyping relation binds unbound type variables to the sub- or supertype they are compared to. Casts can only be inserted for types without type variables. No coercions can be used to a type that has an unbound type variable or a type variable from the type scheme of the definition. The latter restriction is due to type erasure in the runtime of Stratego.

Parametricity. Typical Stratego code can inspect any term with a match strategy or a rewrite rule. This means that, if allowed, Stratego code could inspect terms that are polymorphic in the strategy type. This would violate parametricity, the guarantee that terms of polymorphic type cannot be deconstructed, but it fits the style of Stratego programs rather well. Therefore we chose not to guarantee parametricity in the type system. Type variables from the type scheme of the definition receive special treatment, where they behave as ? throughout the definition, since the definition can be instantiated with dynamic types for the type variables.

```
step(|state): IADD() -> <pop(|2); push(|r); next> state
where
  <top(|2)> state => [Int(v1), Int(v2)]; <addS> (v2, v1) => r
```

Listing 4.13 An example of a step rule in the Jasmin interpreter.

Type Preserving. The special type preserving type is an important piece of the puzzle to type generic traversals and reduce the number of casts inserted around generic traversals. `TP` may seem to be simply `a -> a`, but there are three key differences. The first is that `TP` provides a limited form of higher-rank types. The second is that while parametrically polymorphic strategies may be called with terms that are dynamically typed and possibly ill-formed, a type preserving strategy must be called with a statically typed term. The third is that a type preserving strategy must either return the given term, call a type preserving strategy on it, or inspect the term with a pattern match or type test. In other words, we cannot call a typed strategy with the input term unless the type of the term has been inspected. We do not insert casts to assert its type as we do for polymorphic strategies, as a type preserving, heterogenous strategy should be useable in a generic traversal that tries to apply the strategy everywhere in a tree. When the strategy is not applicable, it should produce a match failure, not a cast error, otherwise the generic traversal mechanism does not work.

4.5 Evaluation

By design of the type system, the following requirements from Section 4.3.1 were met: existing programs are accepted without annotations, generic traversals are supported in both statically and dynamically typed code, we only needed to add minimal (top-level) type annotations which preserves Stratego's concise style, and the types can be modularly checked for integration with the incremental Stratego compiler.

In this section we evaluate the requirement that we can migrate code from untyped to typed. To do so, we have implemented a stand-alone prototype type checker for Stratego, and used it to type-annotate the pre-existing Stratego code of a Spoofax programming language project. This project is Jasmin, a Java Byte Code assembler language (Meyer and Downing 1997). In particular, the project implements the JasminXT version of the language as defined by Reynaud (2006). The Jasmin language project is used in a compiler construction course, where students transform MiniJava to Jasmin, and then use the Jasmin definition to compile to bytecode files. As a result, the part of the codebase that does compilation is well-exercised. Another part of the codebase, a Jasmin interpreter written in Stratego, is not used in the course. The interpreter defines signatures for a state of the JVM and a step rewrite rule that rewrites the state based on a Jasmin instruction. A program is then a list of instructions, which can be executed over the empty state. Listing 4.13 shows a simple step rule that does integer addition by manipulation of the JVM operand stack and addition provided by Stratego.

```
top(|n) = ?JBCState([JBCFrame(_, _, _, JBCStack(<take(|n)>), _)|_], _)
top = top(|1)
```

Typed and corrected to:

```
top(|int) :: State -> List(Value)
top(|n) = ?JBCState([JBCFrame(_, _, _, JBCStack(<take(|n)>), _)|_], _)
top :: State -> Value
top = top(|1); \[head] -> head\
```

Listing 4.14 A bug found by adding types and inspecting errors. The `top` strategy with term argument gives the first `n` values on the stack, whereas the top strategy without argument *should* give the top value (given how it was consistently used) but instead gave a singleton list with the top value.

The Jasmin codebase contains 3253 lines of manually written Stratego code divided over 36 files. We do not count blank lines and comments, nor generated constructor signatures from the grammar. There are 49 manual definitions of constructor signatures for intermediate representations, and 146 named strategies/rules (counted by unique name).

Changes and Bugs. During our evaluation we added 74 type signatures of standard library strategies that are used in the code, and 117 type signatures to the code of the project. In general, we find that the compilation of Jasmin (the well-exercised part) is free of type errors and is mostly written with a static typing discipline. Most type-annotated strategies (85 out of 117) can be given a static type without use of the dynamic type. Some of the manually defined constructors do not have correct types which was not checked before.

The interpreter clearly has not seen testing. We found numerous type-related bugs, such as inconsistent use of integers and strings containing integers, a pattern match one constructor too shallow for the case that was handled, and the use of the wrong strategy which gives back a list instead of an element from the list. Another example is given in Listing 4.14. Most strategies and rules were written with static type discipline in mind. This part did have some overloading (which could be split for static typing) as well as a messy combination of constructors from the Jasmin AST and a newly introduced intermediate representation without a combining supertype. The interpreter 'step' rewrite rule also takes explicitly ill-formed terms, variants of the defined terms where all strings with numbers are replaced by integers.

Interpretation and Conclusion. We conducted a small experiment that shows that using the gradual type system is useful to guide the transition to a system with a better type discipline. In a codebase which was written with some type discipline in mind, this was an exercise in understanding the existing code and adding type annotations. In some cases a small refactoring was easy enough to do and resulted in well-typed strategies. In the process we found that previously added type annotations helped us find the right annotations for the next strategy, and the type system helped us find some mistakes made when originally guessing the type annotation for a strategy. This reassured us that we were indeed adding a consistent set of type annotations to the code.

A noteworthy exception to the smooth experience was the interpreter 'step' rewrite rules. This part of the code was written on an intermediate representation that looked very close to the defined constructors, but with some amount of desugaring and preprocessing. This intermediate representation was close to the original representation, but with small changes consistently applied on a larger number of constructors. Writing the signatures for this would be tedious, and we did not do this as part of the case study. We are not yet sure if this situation is a good argument for our gradual types, a programming style we should discourage in future Stratego code, or if we should add language support for defining these types of intermediate representations more easily.

4.6 Related Work

Dynamically Typed Strategies. The ELAN language introduced rewrite systems with labeled rules that could be invoked from strategies (Borovanský et al. 1998). Luttik and E. Visser (1997) extended these strategies with modal operators (all, one) for generic traversal. E. Visser, Benaissa and Tolmach (1998) used this as the basis for the design of Stratego. E. Visser and Benaissa (1998) define System S, a core language with operational semantics for rewriting, with matching and building as primitive operations. E. Visser (1999) demonstrates the use of strategies for *strategic pattern matching* to dynamically check the (type) structure of terms, e.g. to recognise subsets of a type schema.

Stratego is applied in the Stratego/XT (Bravenboer, Kalleberg et al. 2008) transformation tool suite and the Spoofax language workbench (Kats and E. Visser 2010) for the definition of program normalisation (Bravenboer and E. Visser 2004), type checkers (Hemel, Groenewegen et al. 2011), program analyses (Bravenboer, van Dam et al. 2006), code generators (Kats, Bravenboer and E. Visser 2008), and more. Chapter 3 introduced an incremental compiler for Stratego.

Erdweg, Vergu et al. (2014) introduce typesmart constructors, smart constructors that dynamically check type-correctness of constructed terms. They integrate support for typesmart constructors in the Stratego runtime. Their approach is all-or-nothing, requiring all intermediate values to be well-typed, which precludes the dynamic typing scenarios of Section 4.2.

Statically Typed Strategies. Typing strategies was pursued by Lämmel and J. Visser (2002) in the context of the Strafunski Haskell library for generic programming inspired by Stratego (Lämmel and J. Visser 2003). They identified the concepts of type preserving and type unifying strategies. Lämmel (2003) formalises these with the `TP` and `TU`(b) types in a type system for System S (E. Visser and Benaissa 1998). We have adopted the `TP` type, but have opted to model type unifying strategies with type dynamic as `? ->` b. SYB is a design pattern encoding the statically typed fragment of Stratego's strategies in Haskell (Lämmel and Jones 2003), using type classes and higher-rank polymorphic functions to provide type transformations (type-preserving) and queries (type-unifying) on arbitrary data, made usable by deriving instances of the type classes automatically.

Gradual Types. The term gradual types was introduced by Siek and Taha (2006) in a paper that adds optional type annotations to simply-typed lambda calculus, by an orthogonal extension with a *consistency* relation. This work kicked off a new line of research in adding gradual types to many different type systems and languages, entirely too many to enumerate here. As noted earlier in the paper, we took the work of Herman, Tomb and Flanagan (2010) to heart and applied it in our type system. This work introduces proxies as special closures that can be introspected at run-time to add more coercions to the input and output. This solves a space leak from adding normal closures with casts around a function when it is passed back and forth between statically and dynamically typed higher-order functions.

Xie, Bi and Oliveira (2018) describe a gradual, higher-kinded polymorphic type system that guarantees parametricity, and manages to keep gradual types orthogonal from subtyping induced by polymorphic functions. This work gave us confidence that our type system with limited higher-kinded polymorphism without parametricity (TP) would be possible too. We took inspiration from their notation for algorithmic typing rules. We differ from the research into gradual types in functional type systems, which typically have a separate consistency relation for gradual types which is orthogonal to the rest of the type system. Our approach to subtyping and cast computation is very close to the pessimistic and optimistic subtyping of Muehlboeck and Tate (2017).

Dynamic Rewrite Rules. In this chapter we have not considered the extension of Stratego with *scoped dynamic rewrite rules* (Bravenboer, van Dam et al. 2006), which are used to define context-sensitive transformations such as type checking (Hemel, Groenewegen et al. 2011), function inlining (E. Visser 2001a), and transformation based on data-flow analysis (Olmos and E. Visser 2005). Such rules are dynamic in the sense that new rule instances are added at run-time. With respect to static typing, such dynamic rules can be type checked with the type system from this chapter. When dynamic rules have type dynamic their outputs may be checked at run-time. When dynamic rules have a static type annotation their definitions and applications are checked like regular rewrite rule definitions. So, while the behaviour of dynamic rules is dynamic their typing is static.

4.7 Conclusion

We have introduced the design of a gradual type system for Stratego with a series of idiomatic examples, and presented a formal definition of the type system for Core Gradual Stratego. This type system can statically type many stratego programs, including type preserving and type unifying generic traversals. The gradual types support partially dynamically typed Stratego programs and provide a migration path for existing dynamically typed Stratego code.

To evaluate the implementation of our type checker, we demonstrated this migration path on an existing Stratego codebase, and find it works rather well. What is left to future work is investigating whether we need more language support for easily defining the types of intermediate representations, and to experimentally evaluate the overhead of the casts that our type checker inserts. We also hope to track more properties statically, such as partiality of strategies and boundedness of variables, and use all this static information for optimisations in the back-end of the compiler.

A final word of warning for those who wish to implement a type checker for gradual types: The silent insertion of casts can easily hide bugs in your type system when you write small programs for exploratory testing. So test profusely, including the results after cast insertion.

Chapter 5

Optimising First-Class Pattern Matching

Abstract. Pattern matching is a high-level notation for programs to analyse the shape of data, and can be optimised to efficient low-level instructions. The Stratego language uses *first-class pattern matching*, a powerful form of pattern matching to which traditional optimisation techniques cannot be applied directly.

In this chapter, we investigate how to optimise programs that use first-class pattern matching. Concretely, we show how to map first-class pattern matching to a form close to traditional pattern matching, on which standard optimisations can be applied.

Through benchmarks, we demonstrate the positive effect of these optimisations on the run-time performance of Stratego programs. We conclude that the expressive power of first-class pattern matching does not hamper the optimisation potential of a language that features it.

Based on: Jeff Smits, Toine Hartman and Jesper Cockx (2022). 'Optimising First-Class Pattern Matching'. In: *Software Language Engineering (SLE), Proceedings*, pages 74–83. DOI: 10.1145/3567512.3567519

5.1 Introduction

Pattern matching is a fundamental tool for expressing programs by case analysis, and is used in functional programming, term rewriting, logic programming, and more.

In many languages, pattern matching is expressed through *case expressions*, which combine matching, variable binding, and control flow in a single construct. In contrast, *first-class pattern matching* (E. Visser and Benaissa 1998) breaks apart these concepts into three separate constructs.

For example, Listing 5.1 shows a case expression in OCaml, and the same expression in Stratego (E. Visser 2001b; Bravenboer, Kalleberg et al. 2006; Bravenboer, Kalleberg et al. 2008) using first-class pattern matching. The *match expression* `?pattern` matches the pattern against an implicit scrutinee, the 'current term', and either continues with pattern variables bound or fails. *Scoping* `{n* : S}` binds the variables `n*` locally for use in patterns and expressions. Finally, the *choice operator* `fst <+ alt` runs the first argument and tries the alternative in case it fails. Because Stratego has these separate concepts, backtracking from a failed pattern match is a core feature of the semantics.

Programs that use case expressions can be compiled into a decision tree (Augustsson 1985; Maranget 2008) or an automaton (Gräf 1991; Maranget 1994; Nedjah, Walter and Eldridge 1997; Fessant and Maranget 2001), which can be optimised by reordering branches and arguments. However, these techniques do not apply directly to first-class pattern matching, as matches are spread out in the program. As a consequence, they are not used in Stratego, hurting the run-time performance of Stratego programs.

Our main contributions are the following:

- We demonstrate the mismatch between common pattern match optimisations and first-class pattern matching by example (Section 5.3).
- We describe an intermediate representation (IR) for Stratego that resembles case expressions, and show how to transform Stratego programs to this IR (Section 5.4).
- We evaluate our IR and optimisation with a prototype implementation in the Stratego compiler, benchmarking to two different workloads (Section 5.5).

We starts with an introduction to Stratego in Section 5.2.

5.2 A Short Introduction to Stratego

Stratego (E. Visser, Benaissa and Tolmach 1998) is a gradually typed (see Chapter 4) term rewriting language with programmable rewrite strategies. The terms that are rewritten are ATerms (van den Brand, de Jong et al. 2000), whose sorts and constructors can be defined in Stratego. There is an open world assumption, i.e. Stratego assumes that there may be more constructors and sorts than are defined.

Stratego has syntactic sugar over a core language that strongly resembles System S (E. Visser and Benaissa 1998), a core calculus for rewriting and strategies. The Stratego core grammar is shown in Figure 5.1 and defines identity and

```
let calc n = match n with
  | Plus  (S m, n)   -> S (Plus (m, n))
  | Minus (n,   O)   -> n
  | Minus (S m, S n) -> Minus (m, n)
  | Minus (O,   S _) -> failwith "oops";;

Calc = {m, n: ?Plus( S(m), n    ); !S(Plus(m, n)) }
    <+ {n   : ?Minus(n    , O() ); !n             }
    <+ {m, n: ?Minus(S(m), S(n)); !Minus(m, n)   }
    <+          ?Minus(O() , S(_)); fatal-err(|"oops")
```

Listing 5.1 A case expression in OCaml (top) and Stratego (bottom) with four branches, demonstrating matching, scoping, and choice operators in Stratego.

$S ::= \texttt{id} \mid \texttt{fail}$	identity and failure
$\mid S \texttt{ ; } S$	sequence
$\mid \texttt{\{ } \overline{n}\texttt{: } S \texttt{ \}}$	scope
$\mid \texttt{?}T \mid \texttt{!}T$	match and build
$\mid S \texttt{ < } S \texttt{ + } S$	guarded choice
$T ::= n$	variable
$\mid l$	literal
$\mid n(\overline{T})$	constructor

Figure 5.1 Stratego core grammar.

failure strategies, sequences, lexical scopes for term variables, match and build, and guarded choice. Matching and building is done on term patterns, which includes variables, literals, and constructors and their arguments.

The semantics of these strategies and terms is defined in Figure 5.2 on page 119. For a full explanation of these rules see the System S paper (E. Visser and Benaissa 1998). We will only go over the highlighted rules that pertain to first-class pattern matching. Each rule takes a pair of the store and the 'current term', where the store keeps local variable bindings to values. The rule applies the strategy on the arrow to produce another pair of store and term. The result on the right-hand side of the arrow may also be failure (the $\uparrow$).

The first two highlighted rules describe scoping behaviour. A list of fresh variables of a scope are removed from the store for execution of the body of the scope, then bindings of those variables from before the scope ($St|\overline{x}$) are added again, while preserving other bindings from the body of the scope. Failure in the scope body is propagated.

The next three rules are part of the semantics for matching, in particular matching a variable. An unbound variable is bound in the store. A *bound* variable's binding is compared against the current term, failing if they differ. This provides for non-linear pattern matching semantics in Stratego.

The final four rules are for the guarded choice: Whether the guard (first strategy) fails decides which of the second and third strategy is evaluated. This can result in local backtracking of variables: In the last rule, the failed guard s_1 causes s_3 to be evaluated on the original store St, without bindings from the partial execution of s_1.

Our main point here in looking at the semantics of the core of Stratego, is how pervasive first-class pattern matching is to a language design. Because of the use of backtracking, every construct of the language needs to take 'failure' into account. At the same time backtracking provides the opportunity for a different code style in Stratego than in a typical functional programming language. An alternative result ('fail') is always available, without the requirement to explicitly propagate that failure, as it freely bubbles up to the nearest point where a choice catches it.

Separate matching and scoping have their own benefits. A typical use case is found in the Stratego standard library strategy `fetch-elem(s)`. This strategy returns the element in a list for which the strategy parameter `s` matches. It can be implemented as follows:

```
fetch-elem(s) =
  is-list; one(s; ?x); !x
```

What this code does is (1) test that the input term is a list, (2) use a generic traversal primitive[1] to visit the list elements, attempting to apply the strategy parameter until it succeeds on one of the list elements, and (3) build variable `x` to return its binding as the result of `fetch-elem`. Notably, the strategy parameter to the generic traversal attempts to apply `s` and if it succeeds it binds the result to `x`. The resulting list is ignored as we have found what we wanted during the traversal and bound it to a variable that is scoped outside of the traversal by `fetch-elem`. This trick can be applied to much deeper traversals than a single list, allowing easy extraction of information without result tuples everywhere.

5.3 Pattern Matching Optimisation and First-Class Pattern Matching

The naive way to execute pattern matching of a case expression would be to attempt each branch of the case expression in isolation. For example, when executing the OCaml `calc` function from Listing 5.1 on `Minus (S O, S O)`, the naive method would: (1) fail to match the first pattern because the outermost constructor does not match, then (2) attempt to match the second pattern, match the outermost constructor, but fail to match on the second argument, then (3) attempt to match the third pattern, matching the outermost constructor *again*, which already demonstrates the naivety. Each pattern is tested in isolation, even when we know the outmost constructor from the previous branch. Pattern matching optimisation techniques leverage the information gathered by previously tried branches to make pattern matching faster.

In Stratego the execution method for the code in Listing 5.1 is the naive method described just now. Even if we wrote more high-level rewrite rules that looked more like the OCaml code, it would still desugar to the Stratego code in Listing 5.1. And the desugared code definitely tests each pattern in isolation during a separate first-class pattern matching operation.

[1]This is a central feature of Stratego for its programmable rewrite strategies, for more on this, see E. Visser and Benaissa (1998, § 2.3).

Strategy S applied to store St and term T $\boxed{St;T \xrightarrow{S} St;T}$

$$St;t \xrightarrow{\textbf{id}} St;t \qquad St;t \xrightarrow{\textbf{fail}} \uparrow$$

$$\frac{St;t \xrightarrow{s_1} St';t' \quad St';t' \xrightarrow{s_2} St'';t''}{St;t \xrightarrow{s_1;s_2} St'';t''} \qquad \frac{St;t \xrightarrow{s_1} St';t' \quad St';t' \xrightarrow{s_2} \uparrow}{St;t \xrightarrow{s_1;s_2} \uparrow} \qquad \frac{St;t \xrightarrow{s_1} \uparrow}{St;t \xrightarrow{s_1;s_2} \uparrow}$$

$$\frac{St \setminus \overline{x};t \xrightarrow{s} St';t'}{St;t \xrightarrow{\{\overline{x}:s\}} (St' \setminus \overline{x}) \cup (St \mid \overline{x});t'} \qquad \frac{St \setminus \overline{x};t \xrightarrow{s} \uparrow}{St;t \xrightarrow{\{\overline{x}:s\}} \uparrow}$$

$$\frac{x \notin Dom(St)}{St;t \xrightarrow{?x} St \cup \{x \mapsto t\};t} \qquad \frac{St(x) = t}{St;t \xrightarrow{?x} St;t} \qquad \frac{St(x) \neq t}{St;t \xrightarrow{?x} \uparrow}$$

$$\frac{St_0;t_1 \xrightarrow{?t'_1} St_1;t_1 \ldots St_{n-1};t_n \xrightarrow{?t'_n} St_n;t_n}{St_0;f(t_1,\ldots,t_n) \xrightarrow{?f(t'_1,\ldots,t'_n)} St_n;f(t_1,\ldots,t_n)} \qquad \frac{f \neq g \vee m \neq n}{St_0;g(t_1,\ldots,t_m) \xrightarrow{?f(t'_1,\ldots,t'_n)} \uparrow}$$

$$\frac{t = l}{St;t \xrightarrow{?l} St;t} \qquad \frac{t \neq l}{St;t \xrightarrow{?l} \uparrow}$$

$$\frac{\text{vars}(t_2) \subseteq Dom(St)}{St;t_1 \xrightarrow{!t_2} St;St(t_2)} \qquad \frac{\text{vars}(t_2) \nsubseteq Dom(St)}{St;t_1 \xrightarrow{!t_2} \uparrow}$$

$$\frac{St;t \xrightarrow{s_1} \uparrow \quad St;t \xrightarrow{s_3} \uparrow}{St;t \xrightarrow{s_1 < s_2 + s_3} \uparrow} \qquad \frac{St;t \xrightarrow{s_1} St';t' \quad St';t' \xrightarrow{s_2} St'';t''}{St;t \xrightarrow{s_1 < s_2 + s_3} St'';t''}$$

$$\frac{St;t \xrightarrow{s_1} St';t' \quad St';t' \xrightarrow{s_2} \uparrow}{St;t \xrightarrow{s_1 < s_2 + s_3} \uparrow} \qquad \frac{St;t \xrightarrow{s_1} \uparrow \quad St;t \xrightarrow{s_3} St';t'}{St;t \xrightarrow{s_1 < s_2 + s_3} St';t'}$$

Figure 5.2 Stratego core operational semantics. ↑ is failure. Some rules for first-class pattern matching are highlighted.

Note that first-class pattern matching is not an intermediate representation used by compilers or publications for the description of Stratego's semantics, this is part of the language and actually used by the end user. A common pattern in Stratego is to have a ruleset where some rewrite rules match a different pattern but have otherwise the same logic and result. To remove this code duplication, we can use the core operations like match, choice, and build to write a single strategy that matches both patterns. For example, if we compute the pessimistic time complexity of a language with normal and parallel for loops we see some code duplication:

```
max-complexity: For(i, lo, hi, b) -> <subtS> (hi, lo)
max-complexity: ForPar(i, lo, hi, b) ->
                  <subtS> (hi, lo)
```

We can remove this code duplication using first-class pattern matching:

```
max-complexity =
    (?For(i, lo, hi, b) <+ ?ForPar(i, lo, hi, b))
  ; <subtS> (hi, lo)
```

In general, we cannot expect all Stratego code to match case expression style code as closely as the example in Listing 5.1. In other words, the information that is close together in a match case expression, can be farther apart in Stratego code, and not readily available to fuel optimisation. In this chapter, we will tackle the general problem of optimising first-class pattern matching.

5.4 A Stratego Compatible Case Expression

Pattern match optimisation is a well researched problem in the context of functional programming and case expressions (Cardelli 1984; Baudinet and D. MacQueen 1985; Augustsson 1985; Schnoebelen 1988; Gräf 1991; Sekar, Ramesh and Ramakrishnan 1995; Nedjah, Walter and Eldridge 1997; Maranget 2008). The key idea of our work to find a comparable construct that interacts well with Stratego's backtracking semantics, and translate first-class pattern matching to that construct. Once we have a clear list of branches, we can reuse previous pattern match optimisation ideas, adapting them to Stratego's execution model with local backtracking.

We first look at Stratego code that we can easily translate, to find out what minimum requirements there are for our case expression construct in Stratego. Listing 5.1 shows an already desugared form of rewrite rules. This core Stratego code is a regular structure of a chain of choices, where each guard is a scoped expression starting with a match. These matches should become the left-hand sides of our case expression branches. Our construct also needs to be able to introduce variables like the scopes, have right-hand sides for the builds, and guards in case anything fails and other (overlapping) patterns need to be tried. An example of the case expression that complies with these requirement is displayed in Listing 5.2.

Each branch has some local variables to scope, a pattern, a right-hand side, and optionally a guard. As you can see, the right-hand side of the last case statement is the identity strategy. All the logic is put into the guard of the

```
Calc = match sequential
  case m, n | Plus(S(m), n): S(Plus(m, n))
  case n    | Minus(n, O()): n
  case m, n | Minus(S(m), S(n)): Minus(m, m)
  case      | Minus(O(), S(_))
       when fatal-err(|"Negative result"): id
end
```

Listing 5.2 A case expression version of Listing 5.1.

```
match sequential                              {xs: ?t_0; s_1 < s_2 +
  case xs | t_0                        ⇒      match sequential
      when s_1: s_2                             // more cases
  // more cases                               end }
end

match sequential
  // empty                             ⇒      fail
end
```

Listing 5.3 Naive semantics for case expressions by transformation to Stratego core.

match, where, if anything fails, we can backtrack to another case. This is a Stratego interpretation of the concept of guards, where failure rather than a boolean value governs the behaviour. In later sections we will see how these guards play a key role in translating first-class patterns to case expressions despite Stratego's backtracking semantics.

The naive semantics of our case expression can be shown through a translation back into Stratego core shown in Listing 5.3. A pattern match on term `t_0` and guard `s_1` decide whether we evaluate RHS `s_2`. It is naive because evaluation on e.g. `Plus(O(), O())` would still try each pattern and fail every, whereas the ideally we would fail to match in just two steps, matching the outer `Plus` constructor and finding the `O()` as the first child. For the optimisation of case expression patterns, we can exploit overlap and mutually exclusive patterns to get this behaviour. This can be visualised as a *matching automaton*, exemplified in Figure 5.3. Each intermediate node is labeled with the path into the term that is matched. The Λ is the empty path denoting the root term, and $p.n$ is the (0-indexed) n^{th} child of the term denoted by the path p. Each edge is labeled with the constructor or value matched against. The final nodes of the automaton refer to the n$^{\text{th}}$ branch of the case expression.

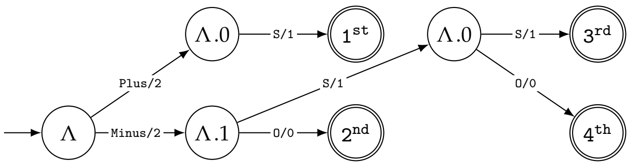


Figure 5.3 Optimised match automaton for the running example.

```
                                          match sequential
s1 < s2 + s3                        ⇒       case | _ when s1: s2
                                            case | _ when s3: id
                                          end
```

Listing 5.4 Transformation of guarded choice to case expression.

```
case xs | _ when {ys: ?p; s1}: s2      ⇒   case xs, ys | p when s1: s2
```

Listing 5.5 Extracting a match from a case expression guard.

5.4.1 Translation into Case Expressions

In order to have a small set of transformations, we defined our translation on Stratego core. While our examples are edited for legibility, Stratego core as generated by the compiler through desugaring makes every variable globally unique and explicitly scoped. We will therefore not concern ourselves with name conflicts and capture-preserving substitution for this translation.

Choices to Cases. The simplest transformation of guarded choices to case expressions is given in Listing 5.4. Without knowing anything about the strategies used in the choice, we can safely put `s1` into the guard for the first branch and `s2` into the right-hand side. `s3` ends up in the second branch, where either the guard or the right-hand side would work. We choose the guard here because it interacts better with the next transformations.

Match Extraction. In order to take advantage of optimisations for case expressions, we need to have patterns in each branch of course. So once we have case expressions, we can start extracting patterns from the guards, as shown in Listing 5.5. Here we have the most general case where a branch already scopes variables `xs`, and the guard scopes variables `ys`, which are combined in the result. As long as the guard then starts with a match, while the branch pattern is a wildcard, we can move the pattern from guard to branch. If anything occurs after that in the guard (`s1`), that stays. And the RHS (`s2`) stays as well, it is not affected by this transformation.

RHS Extraction. Apart from identifying the pattern match in the guard, we can see if the guard is redundant. If we can statically guarantee that the guard always succeeds, it might as well be put into the RHS. A simple local analysis of the guard is enough to find the typical simple cases, such as the build of a term where all variables in the term are guaranteed to be bound.

Flattening. Since guarded choices are a nested structure, our case expressions are also nested in the guards of case expressions at this point. In order to flatten this structure, we define a transformation that flattens a case expression in the guard of another case expression in Listing 5.6. The local strategy `s` that is introduced provides some amount of code sharing, at the expense of creating and calling a closure. This can be avoided by treating these local strategies specially as part of the shared tail in the matching automaton during optimisation of the case expression, although our current prototype does not do so.

```
match sequential
  /* other branches */
  case xs | _ when
    match sequential
      case ys | p1 when s1: s2
      case zs | p2 when s3: s4
      /* more branches */
    end: s5
end
```

⇒

```
let s = s5
in
  match sequential
    /* other branches */
    case xs, ys | p1 when s1; s2: s
    case xs, zs | p2 when s3; s4: s
    /* more branches with extended
   guard and RHS s */
  end
end
```

Listing 5.6 Transformation of nested case expressions to a flat case expression.

5.5 Evaluation

To evaluate our pattern match optimisation, we have implemented a prototype optimisation and included it into a fork of the Stratego compiler. This prototype is limited to generating decision trees rather than automata, and therefore generates duplicate code. But it should still result in improved run time performance. To evaluate this optimisation, we benchmark some Stratego program executions with and without the optimisation. Before we test the performance, we first test the implementation to be behaviour preserving. At the end of this section we consider threats to the validity of our results. We aim to answer the following questions:

RQ1. Does the optimisation improve run time performance of Stratego programs?

RQ2. Is there overhead for small matches?

RQ3. Is the optimisation effective in practice on “normal” code bases?

RQ4. What is the relative compile time cost of the prototype optimisation?

5.5.1 Correctness

For behaviour preservation testing, first we use the compiler test suite and make sure all the tests succeed with optimisation on. This compiler test suite numbers 159 tests and was also used in the past to migrate the compiler backend from C to Java without breaking any edge cases in program behaviour. Then, we added 14 tests that explicitly cover situations that are affected by our optimisation. These all succeed with our current prototype.

We also run the modified Stratego compiler with and without the optimisation on, on the set of benchmark programs used for performance evaluation below, comparing the computed result.

5.5.2 Performance

For our performance evaluation we describe our benchmark setup, and the subjects on which we run our benchmark before discussing the results for each of the subjects.

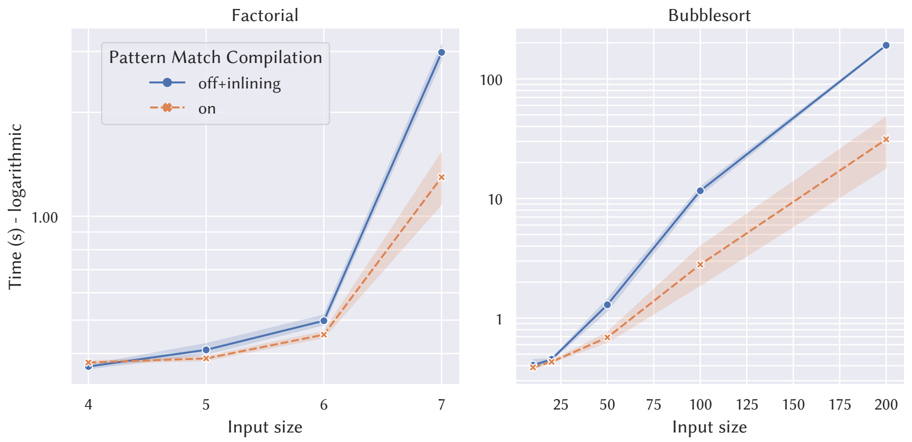


Figure 5.4 Benchmark of REC problems Bubblesort and Factorial.

Environment. We ran our experiments on a Macbook Pro (Early 2013) with an Intel Core i7 2.8 GHz CPU, 16 GB 1600 MHz DDR3 RAM, and an SSD. The machine is running Mac OS 10.14.5, Java OpenJDK 1.8.0_212, and Docker 4.9.1 (81317).

Subjects. We use algorithmic rewriting problems of the Rewriting Engine Competition (REC) (Durán et al. 2010). These problems are generated from the REC language, and can be translated into up to 18 different languages for comparison of rewriting engines. There are problems for sorting (bubble sort, merge sort, quick sort), numeric functions (Fibonacci, factorial, prime sieve of Erastosthenes), and other kinds of problems. The 'algorithmic' problems are expressed as abstract syntax of programs and rewrite rules that do a small-step interpretation of that abstract syntax.

The REC programs are not typical programs as would be written in Stratego by hand, which is acknowledged in Durán et al. (2010, § 4.1). Therefore, like Durán et al., we also use a program-transformation problem set based on the Tiny Imperative Language (TIL) (Cordy 2009).

Data Collection. We use the Java Microbenchmark Harness (JMH) to manage the warm-up of the Java Virtual Machine (JVM), preparing the benchmarks, and repeating runs with different parameters. We used *Single shot time* as the benchmark mode, with 5 warm-up iterations and 5 measurement iterations in two forks of the virtual machine.

REC. Most programs from the REC subject show a significant speed-up, while none slow down due to our optimisation. We show two examples in Figure 5.4, both of which show that we can answer RQ1: our optimisation *does* improve run time performance of Stratego programs. The factorial program is interesting because it has a small match of six different branches, and yet there is a clearly visible run time improvement. Therefore we conclude that, as expected, there is no overhead for small matches (RQ2).

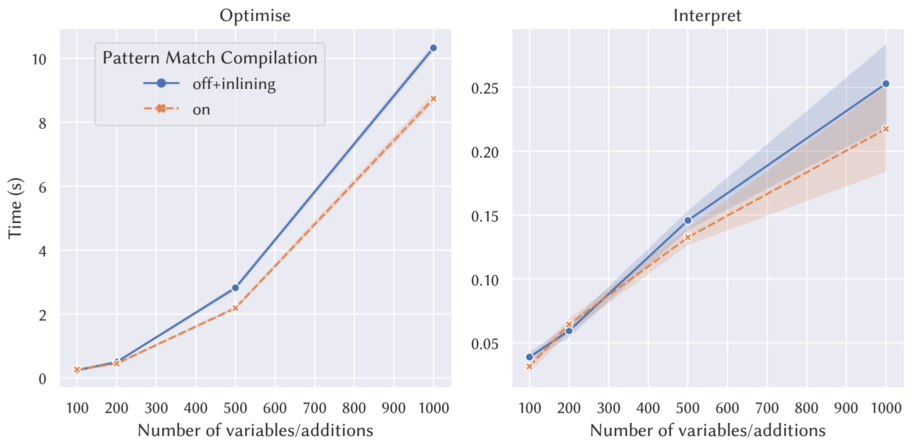


Figure 5.5 Benchmark of TIL optimisation and big-step interpretation on a simple addition program.

TIL. We included TIL for RQ3 to see what impact our optimisation has on a more typical Stratego program that does program-transformation. Figure 5.5 shows that whether we run Stratego code that optimises TIL programs (left) or executes them through big-step transformations (right), there is a small but clear speed-up from our pattern match compiler. This is visible due to our generated large TIL input programs that read one integer, add one to it and put it in a new variable, repeated some number of times, then write the result:

```
var n0; n0 := readint();
var n1; n1 := n0 + 1; // repeated...
write(n500); // ... e.g. 500 times
```

Compilation Time. Figure 5.6 shows that compilation times (RQ4) for REC problems show that our prototype optimisation increases the compilation time of the Stratego compiler, which makes sense as these programs largely consist of rules that the optimisation works on. For the total compile time of the TIL language project, the impact is milder due to smaller rulesets and a larger amount of other compilation work, between 9.68 % and 11.97 % of the median of the baseline. We think that at the current increase of compilation time, the optimisation is worthwhile to include in the main Stratego compiler, with the option to turn it off.

5.5.3 Threats to Validity

We consider generalisability of the results (external validity), factors that allow for alternative explanations (internal validity), and suitability of metrics for the evaluation's goals (construct validity).

External Validity. The results of our evaluation are specific to our implementation for the Stratego language. They are merely a datapoint for the general argument that pattern match optimisation can be used on first-class pattern

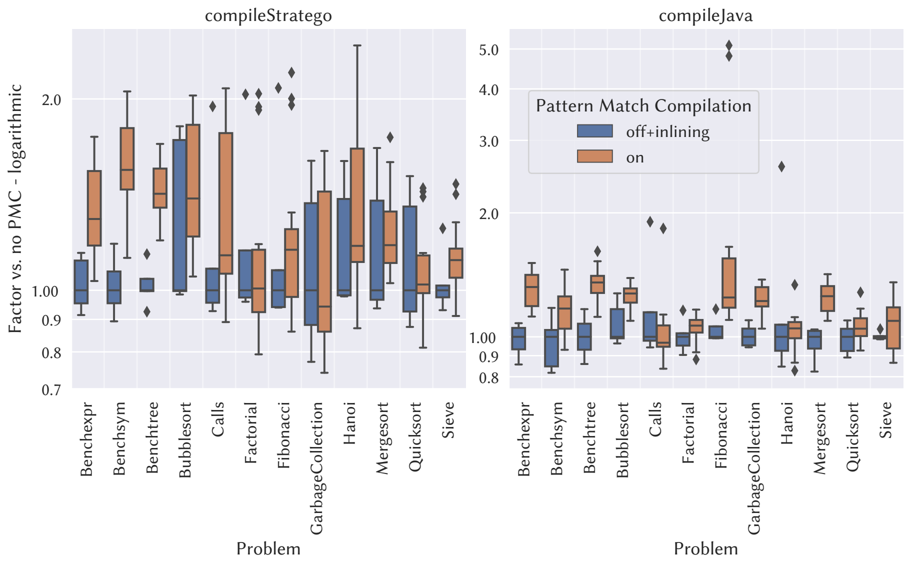


Figure 5.6 Compilation times for REC benchmark problems.

matching. Within Stratego, we have attempted to provide a good range of styles of Stratego programs by using not only the algorithmic problems from REC but also a more typical use of Stratego with TIL.

Internal Validity. A typical alternative explanation for why an optimisation performs well is that it actually removes relevant code and breaks semantics preservation. We have already addressed this concern in Section 5.5.1.

Another explanation would be that we misconfigured our measurements, which we did at one point: we got entirely similar results for our benchmarks because the optimisation was not actually applied in either measurement. We were able to manually inspect the compilation results and notice this though, so we checked that in situations where results are similar.

We used the newest Stratego compiler for our measurements, and our prototype is built on this new compiler. This compiler was designed to be highly incremental, and to achieve this, inlining of strategies was removed. For first-class pattern matching this can be a limitation as a strategy might be defined by choices between differently labeled rewrite rules, but these are optimised separately when not inlined. In our benchmarks we partially mitigated this by manually inlining these code patterns.

Construct Validity. As the experiments are performed in a virtualised environment (Docker), the absolute numbers of our measurements may include some virtualisation overhead. However, the overhead is in all measurements, where we compare between two measurements rather than interpret the absolute numbers. And this allows us to provide a virtual machine image that can be reused to reproduce our experiments.

We used JMH which contains many best practices for benchmarking on the JVM in its default options. We let it run warm-up iterations, checked for stabilisation of the times, and aggregated results to counter noise from background tasks we could not eliminate on the benchmark machine.

5.6 Related Work

Pattern matching has existed since the early ages of computer science, with matching for tree-shaped data being first described in 1972 by Karp, Miller and Rosenberg (1972), with performance being a key consideration from the beginning. It was introduced to the world of programming languages by SASL (Turner 1983) and Hope (Burstall, D. B. MacQueen and Sannella 1980). Since then, it has been one of the staples of functional programming languages like ML and Haskell, and more recently it has been introduced to other mainstream languages such as Python, Swift, Ruby, and Rust.

5.6.1 First-Class Pattern Matching

First-class pattern matching is a feature that so far only exists in the Stratego language (E. Visser, Benaissa and Tolmach 1998; E. Visser 2001b; Bravenboer, Kalleberg et al. 2008; Smits, Konat and E. Visser 2020). The core language of Stratego is System S (E. Visser and Benaissa 1998), which first explored the idea of separating pattern matching into operators *match*, *scope*, and *choice*. E. Visser (1999) describes this unique form of pattern matching in more detail. First-class patterns[2] are a similarly named, but fundamentally different concept to first-class pattern matching.

Cirstea, Lenglet and Moreau (2015) describe a translation from first-class pattern matching to a regular term rewriting system, with the goal of proving termination. This technique could in theory be applied to compile first-class pattern matching by first transforming it into a traditional rewrite system and then applying standard compilation techniques. However, this would likely not lead to good results as the translation has not been designed for this purpose and the encoding might introduce an extra overhead. The technique also seems to require a closed world assumption, while Stratego has an open world assumption.

5.6.2 First-Class Patterns

We consider pattern matching a first-class *expression* in Stratego because it is a primitive operation in the (strategy) expression language. While it sounds similar, first-class patterns are a very different concept, namely that patterns are *values* that can be manipulated in a programming language. This idea shows up in both the (pure) object-oriented programming world (Homer et al. 2012) and the functional programming world (Tullsen 2000; Jay and Kesner 2009).

While this is a powerful idea, it is not directly related to first-class pattern matching as seen this chapter. Indeed, the path polymorphism of Jay and Kesner (2009) provides something more similar to the generic traversal capabilities of Stratego, a feature we did not focus on this chapter.

[2]`https://hackage.haskell.org/package/first-class-patterns`

Depending on the way that first-class patterns are created and used, it may be possible to optimise the pattern match at run-time. If at run-time the information of the different patterns is not co-located enough for a just-in-time optimisation based on traditional pattern matching optimisation, the ideas from this chapter can be reused to bring such information together. This would be particularly relevant for any programming language with first-class patterns and backtracking semantics.

5.6.3 Pattern Match Compilation

There are two main techniques that are used for compiling pattern matching to efficient code: on the one hand, compiling to a *discrimination tree* (also known as a *decision tree* or *case tree*) as pioneered by Overmars and van Leeuwen (1979) and Hoffmann and O'Donnell (1982), and similar work on *deterministic* automata by Gräf (1991), Pettersson (1992) and Nedjah, Walter and Eldridge (1997), and on the other hand compiling to a *backtracking automaton* as proposed by Augustsson (Augustsson 1984; Augustsson 1985) and further developed by Maranget (1994) and Fessant and Maranget (2001). A backtracking automaton has the advantage that it avoids the code duplication that can occur in the construction of a case tree. However, to avoid this duplication it might need to inspect the same term more than once as a result. In theory, this seems like a classic tradeoff between code size and run-time performance. But in practice the difference is less clear, as noted by Fessant and Maranget (2001):

> However, sophisticated compilation techniques exist that minimise the drawbacks of both approaches. [...] In the absence of a practical comparison of full-fledged algorithms, choosing one technique or the other reflects one's commitment to guaranteed code size or guaranteed runtime performance.

A third approach introduced by Jørgensen (1990) is to use *partial evaluation* to compile definitions by pattern matching in a way that avoids examining or decomposing arguments multiple times, and eliminates code duplication. Sestoft (1996) further develops this idea and specialises the partial evaluator to a more traditional pattern match compiler that produces a discrimination tree. The paper notes that this tree can be further optimised by eliminating duplicate trees through hash-consing and by replacing equality checks with switch statements. In our prototype we use switch statements, while eliminating duplicate trees is still left to do.

Strict vs. Lazy Evaluation. In a language that uses strict evaluation such as Stratego, we are free to inspect the arguments of a function in any order, so there is a lot of room for possible optimisation. This is a major advantage over lazy languages, where changing the order of evaluation might influence the termination of our program and hence room for optimisation is more limited (Maranget 1992; Maranget 1994).

Pattern Match Compilation in Practice. Many programming languages compile pattern matching into a discrimination tree, following the example of the ML compiler (Cardelli 1984; Baudinet and D. MacQueen 1985). Other languages compile pattern matching to a backtracking automaton, for example RML (Pettersson 1999) and OCaml. OCaml also applies several optimisations to the control flow of the generated automata to mitigate the performance impact of backtracking (Fessant and Maranget 2001). Finally, yet other implementations of (mostly lazy) languages see pattern matching as mere syntactic sugar for nested case expression, for example Lazy ML (Augustsson 1984; Augustsson 1985) and GHC. To deal with overlapping cases, Lazy ML uses a **default** construct that triggers backtracking. In contrast, GHC expands catchall cases and uses *join points* in its core language to avoid duplication of terms (Eisenberg and GHC development team 2020). In the current Stratego implementation, closures and their calls are computationally expensive, so a direct adaptation of this approach would result in poor performance.

5.6.4 Heuristics

Over the years, a large number of different heuristics have been proposed for producing better discrimination trees or matching automata, depending on whether one wishes to optimise for code size or run-time performance. Detailed comparisons between these different heuristics can be found in the studies by Scott and Ramsey (1999) and Maranget (2008). Here we list some of the most common ones, with an indication of whether they optimise for run-time or code size:

Relevance (run-time and code size) Pick an argument position that is matched on in a higher priority clause (Baudinet and D. MacQueen 1985).

Necessity (run-time) Pick an argument position that must be inspected by *any* matching algorithm (Sekar, Ramesh and Ramakrishnan 1995; Nedjah and de Macedo Mourelle 2001; Maranget 2008).

Large branching factor (run-time) Pick an argument position with the largest number of distinct constructors (Cardelli 1984).

Small branching factor (code size) Pick an argument position with the smallest number of distinct constructors (Baudinet and D. MacQueen 1985; Nedjah and de Macedo Mourelle 2001).

Small arity factor (code size) Pick an argument position where the total arity of all constructors is lowest (Baudinet and D. MacQueen 1985).

Small default (code size) Pick an argument position with the smallest number of wildcard patterns (Baudinet and D. MacQueen 1985).

Scott and Ramsey (1999) find little difference in performance between these heuristics for most practical examples, with a few notable exceptions. Maranget (2008) recommends a combination of necessity, small branching factor, and arity, but also notes that the choice depends on the specifics of the language and the expected kinds of pattern matching code. Hence, it would be interesting to experiment with what combination of heuristics is the most suitable for compiling idiomatic Stratego code.

5.7 Conclusion

We have introduced the problem of optimising *first-class pattern matching* as seen in the Stratego programming language, where pattern matching is broken apart into three constructs: match, scope, and choice. To solve this problem, we have developed a behaviour-preserving transformation that combines these three constructs into an intermediate representation that resembles *case expressions* from functional programming, while still adhering to the local backtracking semantics of Stratego. This allowed us to optimise first-class pattern matching with the same techniques used for 'regular' pattern matching.

We still plan to make some practical improvements to our prototype implementation, investigate which heuristics for the discrimination trees work best for typical Stratego code, and investigate the addition of closed types to Stratego and what performance effect this may have on our optimisation.

Nevertheless, our benchmarks have demonstrated that our current prototype can already have a positive effect on the run-time performance of Stratego programs. With a speed-up that increases with the input size, we expect Stratego users will be willing to pay a 10% increase in compile time.

Chapter 6

Conclusion

In order to develop software, we need tools from the domain of Programming Languages (PL). Such PL tools include compilers and interpreters, but also development environments that contain smart program text editors ('editors'), program analysis and manipulation tools ('refactoring tools'), and live program observers ('debuggers'). Innovative PL tools have a large impact because every software developer uses them. But they also have strict requirements, because slow or buggy PL tools are a detriment to software development. Language Workbenches (LWBs) are suites of tools that are specifically designed to create PL tools. This dissertation has shown several improvements to the Spoofax language workbench (Kats and E. Visser 2010) and how those may be applied elsewhere.

6.1 Language Development Cycle(s)

Chapter 1 introduced the Language Development Cycle(s) (repeated here in Figure 6.1), which are the feedback cycles during language development that a LWB facilitates. In order to test a new idea for a programming language, we need a LWB that allows us to express that idea quickly and concisely. By providing a specification mechanism that is high-level and powerful, a LWB can facilitate the first part of language development, where an idea is made concrete through specification. The expressive power of the LWB facilitates the easy and quick specification. Because the specification is high-level, it allows us to reason about the idea more easily, and gives the LWB the opportunity to analyse the specification to give feedback on its internal consistency. All of this gives feedback on the idea we started out with.

Of course high-level specifications may be used to describe powerful ideas that are difficult to implement. A LWB should also provide a means to create an implementation from the specification. To keep the specification and implementation in agreement, these should be compared to each other automatically by the LWB. This can be done by explicit comparison, generation of part of the implementation from the specification, or even by providing

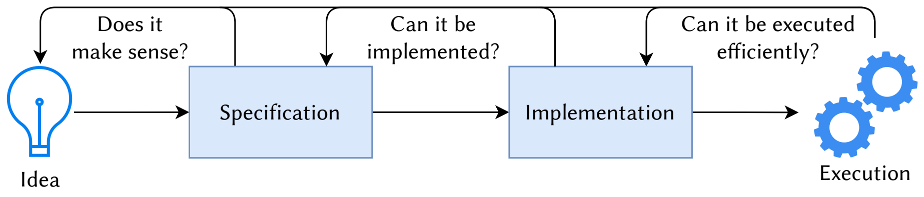


Figure 6.1 The Language Development Cycle(s)

fully executable specifications. Looking for an executable algorithm for a specification can lead to useful changes to the specification of the language, or even to the fundamental idea from which we started.

The final feedback cycle comes from the performance of the implementation. If our new language idea cannot be executed efficiently, it will be difficult to include into useful PL tools. This will inform the way we implement, specify, and even think about our original idea. The challenge for the LWB is to provide the span from high-level specification capabilities all the way to a reasonably efficient implementation capabilities.

6.2 How Language Development Cycles can be Sped Up

The main research question driving the work presented in this dissertation is: *How can the Language Development Cycle be sped up?*
We have looked at different forms of speed here, from specification and development speed through expressive power, to speed and quality of automated feedback on specifications, and implementations, to inform our development, and in terms of execution speed of our implementations.

FlowSpec. In particular, we now have FlowSpec, a new declarative specification language that can describe control-flow and data-flow in a language-parametric way. This new specification language from Chapter 2 provides expressive power, as we can define control- and data-flow in domain terms. The Domain-Specific Language (DSL) provides the opportunity to give error messages on the specification in domain terms as well. These speed up the first cycle in language development, from idea to specification.

Since FlowSpec specifications are executable, the second cycle from specification to implementation is reduced to a minimum, it is as fast as compiling the specification. The compiler targets a runtime system that uses an efficient worklist algorithm, and we have shown with benchmarks how this provides reasonable performance. Therefore the final feedback cycle is also improved by FlowSpec.

Of course, FlowSpec only influences the control- and data-flow part of a programming language specification and the tools derived from that. Stratego has a larger potential as the glue language of Spoofax, in which we can implement anything that we cannot in the other meta-DSLs.

Stratego. To improve the Stratego term rewriting language, we looked at the pain points of using the language. A large pain point for Stratego is its slow compilation time, which frustrates the user that is trying to iterate on an implementation to improve the quality by for example fixing bugs, mismatches with the specification, or performance issues (in other words, the third cycle). Although the Stratego language has features that seemingly require whole-program compilation, we have seen in Chapter 3 that it is possible to have an incremental Stratego compiler that is backward compatible and reuses most of the original compiler.

To catch bugs faster than compiling and testing, a type system is valuable to avoid type-related bugs. These are common enough in Stratego, and by catching them in the type system we can speed up the second and third language development cycles. But since Stratego is dynamically typed, with powerful generic traversals that work on any data, a strong static type system does not fit. However, a gradual type system does. And in Chapter 4 we have seen that the introduced gradual type system can be used to gradually introduce a type discipline to old code, while providing a proper type system for any new code.

In order to get to PL tools with reasonable performance, we do not only need to catch bugs though. We also need the execution of Stratego programs to be fast enough. Since the performance of Stratego programs leaves something to be desired sometimes, Chapter 5 describes a way to optimise a very common operation in Stratego: pattern matching. There are good solutions from the Functional Programming (FP) world for optimising pattern matching, but these do not apply directly to Stratego's atypical first-class pattern matching. But as we saw, we can find program fragments that map to FP's case expressions form of pattern matching, which allows us to apply existing optimisation techniques.

6.3 Reflections and Future Work

We have looked at strategic improvements to a language workbench in this dissertation. The approach to this research was that of programming systems research, where we develop research software to the point of practically usable tools. While this is a good test for the underlying ideas, there is a cost to such an approach. When you work long enough on research software for it to become practically useful, you are more likely to have to pay back some technical debt. A long living research vehicle like the Spoofax language workbench compounds ideas and insights, but also technical debt that at some point cannot be ignored if you wish to keep building on it. Nevertheless, building research software to the point of practical use provides unique insights into what it takes for an idea to become truly beneficial to the world.

6.3.1 FlowSpec

Developing the FlowSpec DSL was a learning process in programming systems research. Early on, the project was built upon and integrated with some young, experimental systems within Spoofax that were under heavy development. This slowed down the research process more than necessary (depending on your definition of necessary).

The performance of FlowSpec as presented in this dissertation is passable, but ultimately the analysis time will increase with the size of the program. In a setting where small changes are made to the program, such as when a data-flow analysis is used for an editor service, it would be better to have an incremental analysis. In fact, even when we use data-flow analysis to inform optimisation of a program through program transformation, we may want to do this to a fixpoint, alternating between analysis and transformation. This would also benefit from incremental analysis, so the FlowSpec runtime from Chapter 2 has a serious limitation here. Thankfully, I got to supervise a Master's thesis project by Matthijs Bijman, in which he designed and implemented an incremental runtime for FlowSpec (Bijman 2022).

Both the concrete implementation of FlowSpec and the conceptual ideas of the language can be used today, by anyone interested. Any user of the Spoofax language workbench can now extend their language with a declarative definition of control-flow, build data-flow analyses on top of that, and use those analyses to their advantage. Any researcher can take the design ideas behind the FlowSpec DSL, and remix or extend these to create an even better meta-DSL for the declarative specification of control- and data-flow. In the following paragraphs, I present a number of extensions for FlowSpec that I have in mind.

Control Flow. There are many opportunities to increase the expressiveness of the FlowSpec DSL. At the moment, FlowSpec only contains the concept of names to manage shadowing in data-flow analyses, but its use could be expanded for name-dependent control-flow such as labeled breaks. This requires an association of names with parts of the Control-Flow Graph (CFG) or Abstract Syntax Tree (AST). Unlabelled `break` and `continue` statements, as well as `return` statements could be modelled with implicit, lexically scoped names.

The `finally` block of a try-catch (e.g. in Java) is a particularly strange bit of control-flow to model as it inserts itself at the end of the try and catch blocks, and any control-flow that abruptly exits those blocks unless it is an exception in the try block that gets caught by a catch block. Modelling this in FlowSpec required named and parameterised control-flow rules. It would also be interesting to allow the CFG of the `finally` block to be duplicated as we know where it was entered from and where it exits to in each program trace.

Some programming languages under-specify their control-flow within expressions without side-effects[1], which allows them to be executed in any order. If we add this as a concept within control-flow in FlowSpec, we can use it to more accurately specify control-flow of those languages. A major benefit of adding this concept is that we could specify relaxing CFGs in FlowSpec as well: we first build a CFG, use it for a data-flow analysis that finds control and data dependencies between CFG nodes, and finally use the results of this analysis to relax control-flow between sub-CFGs that are independent of each other. Such a relaxed CFG makes it easy to move code around safely and can be used to find optimisation opportunities.

Inter-procedural Analysis. Another limitation to the expressiveness of FlowSpec is that it can only do intra-procedural analysis. Analysis between procedure or function calls was out of scope for this dissertation. The interaction between accuracy of the call graph and the cost of analysis is non-trivial: We can avoid work if we can accurately predict which definition a dynamically dispatched call resolves to, but accurate prediction of that requires an expensive data-flow analysis.

[1]Some languages don't even specify the order of expressions *with* side-effects, but let us ignore such horrors in this instance.

For FlowSpec, I believe that function summaries (Rountev, Ryder and Landi 1999; Schubert, Hermann and Bodden 2021) are a particularly interesting approach to inter-procedural analysis. It would be interesting to see if the DSL allows us to reason about user-implemented data-flow analyses, in order to automatically derive a way to summarise them. Of course the language can be extended to limit the way in which data-flow analyses can be specified if that is necessary for summaries, providing high-level error messages about why the restriction is necessary. An extension could also give the user the ability to tweak the way summaries work, to influence trade-offs in accuracy and efficiency that are language specific.

6.3.2 Stratego

Working on an existing programming language can be challenging because of the constraint of backward compatibility. There are certainly properties of the design of Stratego that can/could be improved. But depending on the problem in question, it could be difficult to actually do something about it, especially if there was discussion on (1) if the property was even a problem in the first place, or (2) what the solution to the undesirable property should be, or (3) how hard it would be to keep the solution backwards compatible. The three chapters related to Stratego tackled problems of different natures, where these discussions led to publishable research. This research has had a positive effect on Stratego, but has also provided insights that are useful beyond Stratego.

Incremental Computing. The incremental Stratego compiler is the product of many constraints to keep the language, output and runtime backwards compatible. In the future, there may be a point where it becomes worthwhile to break backwards compatibility of the runtime to allow better code generation pattern in the compiler. This would provide the opportunity to add separate compilation support for Stratego, simplifying incremental compilation to file-based caching and dependencies.

Nevertheless, our incremental compiler has demonstrated how building an incremental system that is backwards compatible is possible, even when there are severe constraints. This is an insight that I believe benefits the world of software development at large. Existing, very useful software that is too complex and expensive to re-implement, could be made incremental. While the Stratego compiler is merely an exemplar, it shows how different properties of software may influence how we can build in incrementality. This is not an automated process, but is based a detailed analysis of the software architecture. One clear property is that parts of the system that follow a (mostly) pure data transformation approach can easily fit inside an incremental system. Our approach also shows that parts of the system that rely heavily on side-effects may be isolated and kept non-incremental. We can then test if they become a bottleneck in the incremental process, and if necessary consider re-implementing only that part to be more amenable to incremental computation.

Gradual Types. The work on Stratego's gradual type system was started with discussions on what we would want or expect from a type system for Stratego (the idea phase). However, fairly quickly we found that it would be easier to design features of the type system if we saw it in action. We decided not to use the Statix meta-DSL in Spoofax for this because it had significant execution speed issues at the time. Besides, we had learned from FlowSpec not to build on overly new and experimental system for unrelated research.

The prototype implementation of the type system informed our ideas and allowed us to iterate on the design of the type system. The evaluation of the type system on some Stratego projects gave us confidence that our type system implementation would be practically useful.

But the largest missing piece for Stratego's gradual type system is correctness proofs. Although the usual soundness of type systems does not apply to gradual type systems directly, as New, Licata and Ahmed (2021) explain, there are gradual type soundness theorems and the gradual guarantee, where the latter emphasises that 'the gradual migration process should be "smooth"'. I would feel more confident of the consistent behaviour of Stratego's gradual type system if we had a definition of Stratego's gradual type system with a mechanised proof of gradual type soundness and the gradual guarantee and based the implementation on this definition.

Despite that future work, researchers can take our gradual type system as another piece of evidence that gradual types are an interesting and useful tool to bring some of the benefits of static types to a dynamically typed language. It combines the practical benefits of the Stratego language with previous work on statically typed, but restricted, generic traversals. Hopefully, this sparks renewed interest in generic programming, which can be very powerful.

6.3.3 Language Development in Spoofax

Developing research software for and inside of Spoofax has made a positive impact on our ability to create practically usable tools. I believe that Spoofax could be practically useful to more people outside of our research group in Delft. But for that to happen, we would need to advertise Spoofax better, and spend more time polishing what we currently have. Another option would be to make the different meta-DSLs of Spoofax individually usable. Both of those options, and the advertisements, are mostly not research. Spoofax as a research project could use some (budget for) non-research help, as our researchers have too little time to keep up with that work.

Typed AST Generation. There are also more research opportunities within the Spoofax ecosystem. Grammars written in Syntax Definition Formalism (SDF) version 3 currently produce a parser that creates an AST with a generic term interface in Java. If we want to advertise the SDF 3 meta-DSL as a separate product, we should generate Java types that follow the grammar, similar to other tools (Parr and Quong 1995; Cervelle, Forax and Roussel 2006). There is still a design space to explore here, with how we represent ambiguous parse results, are the trees mutable or immutable (Lippert 2012), can we generate specialised generic traversals for the trees?

Generic Traversal Fusion. Speaking of generic traversals, there is a special optimisation in the Stratego compiler that assumes there is an `innermost` generic traversal defined with specific semantics. Based on the assumed semantics, this optimisation fuses the generic traversal and the set of rewrite rules it is called with, essentially putting the traversal inside of the rewrite rules (Johann and E. Visser 2001). It would be interesting to explore whether we can generalise this idea to any generic traversal written in System S, and when such an optimisation would be helpful. Perhaps the same approach of looking at System S primitives would allow us to fuse together multiple traversals into a single one. Fusing traversals has been done before (Sakka, Sundararajah and Kulkarni 2017; Petrashko, Lhoták and Odersky 2017), but by focussing on System S we might be able to find a less restricted form of fusion.

Interactive Interpreters and Debuggers. The Read–Eval–Print Loop (REPL) is an interactive interpreter that accepts program fragments of a programming language, sometimes with slightly adapted syntax to better fit the interactive nature. It would be nice if Spoofax got special support for creating a REPL for a programming language. We would need to optionally define some special grammar symbols particular to the REPL and define the start symbol for that context. The derived parser should handle parsing line-by-line, so that multi-line definitions in the REPL are supported out of the box. The dynamic semantics specification of the language could be used directly to create an interpreter. A particularly interesting question that would come up is what a REPL for a DSL might look like. van Binsbergen et al. (2020) has some insights to get started in this area.

An interpreter would also be a useful feature inside of an interactive debugger. This is another PL tool that Spoofax should gain special support for. As Spoofax generates plugins for different editors, it should also provide the option to plug into those editors' interactive debugging interfaces. The interpreter should be able to step through a program in a programming language defined in Spoofax. The state of the program should be inspectable, for which it can be useful to write a snippet of code in the programming language to inspect that state. This is where the interpreter comes in. As with the interpreter, it would be interesting what an interactive debugger for a DSL might look like.

Names and Types. Before scope graphs, Name Binding Language (NaBL) version 2, and Statix, we had NaBL version 1 and Type System Language (TS) in Spoofax. These older languages were less powerful and did not have a clear, formalised semantics. Still, I yearn for their simplicity sometimes. I would rather read the NaBL 1 code on the left than the Statix equivalent on the right to define the rule for method names and scoping local variables:

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

```
methodOk(s, Method(_, m, _, b)) :-
  {s_mthd s'}
  new s_mthd, !mthd[m, s_mthd] in s,
  new s', s' -LEX-> s, statementOk(s', b).
```

A language inspired by NaBL 1 and TS, that translated to Statix, would provide a gentler learning curve for the specification of names and types in Spoofax.

6.4 Parting Words

The research I have presented in this dissertation improves the feedback cycles in language development. Each piece of research includes software integrated in the Spoofax LWB, and can be used to create the next generation of PL tools. I hope I have inspired others with my work, to apply it in a different context, or to try something in a similar vein. I plan to keep extolling the virtues of Spoofax, while I continue to improve language development in it.

Bibliography

Allen, Frances E. and John Cocke (June 1972). 'A catalogue of optimizing transformations'. In: *Design and Optimization of Compilers*. 1st edition. Automatic Computation, pages 1–30. ISBN: 9780132002042 (cited on page 48).

Association for Computer Machinery (Aug. 2020). *Artifact Review and Badging - Current*. URL: https://www.acm.org/publications/policies/artifact-review-and-badging-current (visited on 9th Nov. 2022) (cited on page 4).

Augustsson, Lennart (1984). 'A Compiler for Lazy ML'. In: *LISP and Functional Programming (LFP), Proceedings*, pages 218–227. DOI: 10.1145/800055.802038 (cited on pages 128–129).

Augustsson, Lennart (1985). 'Compiling Pattern Matching'. In: *Functional Programming Languages and Computer Architecture*. Volume 201. Lecture Notes in Computer Science, pages 368–381. DOI: 10.1007/3-540-15975-4_48 (cited on pages 116, 120, 128–129).

Auslander, Marc A. and Martin Hopkins (1982). 'An Overview of the PL.8 Compiler'. In: *SIGPLAN Symposium on Compiler Construction*, pages 22–31. DOI: 10.1145/800230.806977 (cited on page 16).

Backus, John Warner and William P. Heising (1964). 'Fortran'. In: *IEEE Transactions on Computers* 13.4, pages 382–385. DOI: 10.1109/PGEC.1964.263818 (cited on pages 64, 84).

Baudinet, Marianne and David MacQueen (Dec. 1985). *Tree Pattern Matching for ML (Extended Abstract)*. Technical report. URL: https://smlfamily.github.io/history/Baudinet-DM-tree-pat-match-12-85.pdf (cited on pages 120, 128–129).

Bavelas, Alex (Nov. 1950). 'Communication patterns in task-oriented groups.' In: *The Journal of the Acoustical Society of America* 22.6, pages 725–730. DOI: 10.1121/1.1906679 (cited on page 46).

Bijman, Matthijs (June 2022). 'Dataflow Analysis in a Language Workbench'. Master's thesis. Delft University of Technology. URL: http://resolver.tudelft.nl/uuid:1a32b1ef-524e-48bf-8588-975e0f00e71e (cited on page 136).

Bissyandé, Tegawendé F., Ferdian Thung, David Lo, Lingxiao Jiang and Laurent Réveillère (2013). 'Popularity, Interoperability, and Impact of Programming Languages in 100, 000 Open Source Projects'. In: *Computer Software and Applications Conference (COMPSAC), Proceedings*, pages 303–312. DOI: 10.1109/COMPSAC.2013.55 (cited on pages 1–2).

Borovanský, Peter, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau and Christophe Ringeissen (1998). 'An overview of ELAN'. In: *Electronic Notes in Theoretical Computer Science* 15, pages 55–70. DOI: 10.1016/S1571-0661(05)82552-6 (cited on page 111).

Bravenboer, Martin, Karl Trygve Kalleberg, Rob Vermaas and Eelco Visser (2006). 'Stratego/XT 0.16: components for transformation systems'. In: *Partial Evaluation and Semantics-based Program Manipulation, Proceedings*, pages 95–99. DOI: `10.1145/1111542.1111558` (cited on page 116).

Bravenboer, Martin, Karl Trygve Kalleberg, Rob Vermaas and Eelco Visser (2008). 'Stratego/XT 0.17. A language and toolset for program transformation'. In: *Science of Computer Programming* 72.1-2, pages 52–70. DOI: `10.1016/j.scico.2007.11.003` (cited on pages 52, 65, 71, 90, 111, 116, 127).

Bravenboer, Martin and Yannis Smaragdakis (2009). 'Strictly declarative specification of sophisticated points-to analyses'. In: *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), Proceedings*, pages 243–262. DOI: `10.1145/1640089.1640108` (cited on page 59).

Bravenboer, Martin, Arthur van Dam, Karina Olmos and Eelco Visser (2006). 'Program Transformation with Scoped Dynamic Rewrite Rules'. In: *Fundamenta Informaticae* 69.1-2, pages 123–178. URL: `https://content.iospress.com/articles/fundamenta-informaticae/fi69-1-2-06` (cited on pages 7, 54, 58, 65, 93, 111–112).

Bravenboer, Martin and Eelco Visser (2004). 'Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions'. In: *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), Proceedings*, pages 365–383. DOI: `10.1145/1028976.1029007` (cited on page 111).

Burstall, Rod M., David B. MacQueen and Donald Sannella (1980). 'HOPE: An Experimental Applicative Language'. In: *LISP and Functional Programming (LFP), Proceedings*, pages 136–143. DOI: `10.1145/800087.802799` (cited on page 127).

Cardelli, Luca (1984). 'Compiling a Functional Language'. In: *LISP and Functional Programming (LFP), Proceedings*, pages 208–217. DOI: `10.1145/800055.802037` (cited on pages 120, 128–129).

Cervelle, Julien, Rémi Forax and Gilles Roussel (2006). 'Tatoo: an innovative parser generator'. In: *Principles and Practice of Programming in Java (PPPJ), Proceedings*. Volume 178, pages 13–20. DOI: `10.1145/1168054.1168057` (cited on page 138).

Checkstyle team (2018). *checkstyle – Coding*. URL: `http://checkstyle.sourceforge.net/config_coding.html` (visited on 10th Apr. 2018) (cited on page 16).

Cirstea, Horatiu, Sergueï Lenglet and Pierre-Etienne Moreau (2015). 'A faithful encoding of programmable strategies into term rewriting systems'. In: *Rewriting Techniques and Applications (RTA), Proceedings*. Volume 36. LIPIcs, pages 74–88. DOI: `10.4230/LIPIcs.RTA.2015.74` (cited on page 127).

Combemale, Benoît, Olivier Barais and Andreas Wortmann (2017). 'Language Engineering with the GEMOC Studio'. In: *International Conference on Software Architecture Workshops (ICSAW), Proceedings*, pages 189–191. DOI: `10.1109/ICSAW.2017.61` (cited on page 3).

Cordy, James (2009). *TILChairmarks*. URL: https://www.program-transformation.org/Sts/TILChairmarks.html (visited on 26th July 2022) (cited on page 124).

Danvy, Olivier and Lasse R. Nielsen (2004). *Refocusing in Reduction Semantics*. BRICS Research Series RS-04-26. Department of Computer Science, Aarhus University. URL: https://www.brics.dk/RS/04/26/ (cited on page 6).

Denkers, Jasper, Louis van Gool and Eelco Visser (2018). 'Migrating custom DSL implementations to a language workbench (tool demo)'. In: *Software Language Engineering (SLE), Proceedings*, pages 205–209. DOI: 10.1145/3276604.3276608 (cited on page 90).

De Souza Amorim, Luis Eduardo, Sebastian Erdweg, Guido Wachsmuth and Eelco Visser (2016). 'Principled syntactic code completion using placeholders'. In: *Software Language Engineering (SLE), Proceedings*, pages 163–175. DOI: 10.1145/2997364.2997374 (cited on page 5).

De Souza Amorim, Luis Eduardo and Eelco Visser (2020). 'Multi-purpose Syntax Definition with SDF3'. In: *Software Engineering and Formal Methods (SEFM), Proceedings*. Volume 12310. Lecture Notes in Computer Science, pages 1–23. DOI: 10.1007/978-3-030-58768-0_1 (cited on pages 91, 95).

Durán, Francisco, Manuel Roldán, Jean-Christophe Bach, Emilie Balland, Mark G. J. van den Brand, James R. Cordy, Steven Eker, Luc Engelen, Maartje de Jonge, Karl Trygve Kalleberg, Lennart C. L. Kats, Pierre-Etienne Moreau and Eelco Visser (2010). 'The Third Rewrite Engines Competition'. In: *Workshop on Rewriting Logic and its Applications (WRLA), Revised Selected Papers*. Volume 6381. Lecture Notes in Computer Science, pages 243–261. DOI: 10.1007/978-3-642-16310-4_16 (cited on page 124).

Eisenberg, Richard and GHC development team (May 2020). *System FC, as implemented in GHC*. Technical report. URL: https://gitlab.haskell.org/ghc/ghc/-/raw/97655ad88c42003bc5eeb5c026754b005229800c/docs/core-spec/core-spec.pdf (visited on 14th Feb. 2023) (cited on page 129).

Ekman, Torbjörn and Görel Hedin (2007a). 'The JastAdd extensible Java compiler'. In: *Object-Oriented Programming, Systems, Languages, and Applications (OOPLSA), Proceedings*, pages 1–18. DOI: 10.1145/1297027.1297029 (cited on pages 3, 85).

Ekman, Torbjörn and Görel Hedin (2007b). 'The JastAdd system - modular extensible compiler construction'. In: *Science of Computer Programming* 69.1-3, pages 14–26. DOI: 10.1016/j.scico.2007.02.003 (cited on page 57).

Erdweg, Sebastian, Paolo G. Giarrusso and Tillmann Rendel (2012). 'Language composition untangled'. In: *Language Descriptions, Tools, and Applications (LDTA), Proceedings*, page 7. DOI: 10.1145/2427048.2427055 (cited on page 85).

Erdweg, Sebastian, Moritz Lichter and Manuel Weiel (2015). 'A sound and optimal incremental build system with dynamic dependencies'. In: *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), Proceedings*, pages 89–106. DOI: 10.1145/2814270.2814316 (cited on pages 72–73).

Erdweg, Sebastian, Vlad A. Vergu, Mira Mezini and Eelco Visser (2014). 'Modular specification and dynamic enforcement of syntactic language constraints when generating code'. In: *International Conference on Modularity, Proceedings*, pages 241–252. DOI: `10.1145/2577080.2577089` (cited on page 111).

Fessant, Fabrice Le and Luc Maranget (2001). 'Optimizing Pattern Matching'. In: *International Conference on Functional Programming (ICFP), Proceedings*, pages 26–37. DOI: `10.1145/507635.507641` (cited on pages 116, 128–129).

Flyvbjerg, Bent (Apr. 2006). 'Five Misunderstandings about Case-Study Research'. In: *Qualitative Inquiry* 12.2 (cited on page 4).

Fowler, Martin (1999). *Refactoring - Improving the Design of Existing Code*. Addison Wesley object technology series. ISBN: 978-0-201-48567-7 (cited on page 1).

Fowler, Martin (June 2005). *Language Workbenches: The Killer-App for Domain Specific Languages?* URL: `http://www.martinfowler.com/articles/languageWorkbench.html` (visited on 18th Oct. 2022) (cited on page 1).

Fowler, Martin (2010). *Domain-Specific Languages* (cited on page 2).

Geschke, Charles M., James H. Morris Jr. and Edwin H. Satterthwaite (Aug. 1977). 'Early Experience with Mesa'. In: *Communications of the ACM* 20.8, pages 540–553. DOI: `10.1145/359763.359771` (cited on pages 64, 84).

Ginsbach, Philip, Lewis Crawford and Michael F. P. O'Boyle (2018). 'CAnDL: a domain specific language for compiler analysis'. In: *Compiler Construction (CC), Proceedings*, pages 151–162. DOI: `10.1145/3178372.3179515` (cited on page 60).

Gosling, James, Bill Joy, Guy L. Steele Jr. and Gilad Bracha (May 2005). *The Java Language Specification*. 3rd edition (cited on page 16).

Gräf, Albert (1991). 'Left-to-Right Tree Pattern Matching'. In: *Rewriting Techniques and Applications*. Volume 488. Lecture Notes in Computer Science, pages 323–334. DOI: `10.1007/3-540-53904-2_107` (cited on pages 116, 120, 128).

Groenewegen, Danny M., Zef Hemel, Lennart C. L. Kats and Eelco Visser (2008). 'WebDSL: a domain-specific language for dynamic web applications'. In: *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), Companion*, pages 779–780. DOI: `10.1145/1449814.1449858` (cited on pages 10, 69).

Groenewegen, Danny M. and Eelco Visser (2008). 'Declarative Access Control for WebDSL: Combining Language Integration and Separation of Concerns'. In: *International Conference on Web Engineering (ICWE), Proceedings*, pages 175–188. DOI: `10.1109/ICWE.2008.15` (cited on page 69).

Groenewegen, Danny M. and Eelco Visser (2013). 'Integration of data validation and user interface concerns in a DSL for web applications'. In: *Software and Systems Modeling* 12.1, pages 35–52. DOI: `10.1007/s10270-010-0173-9` (cited on page 69).

Grönniger, Hans, Holger Krahn, Bernhard Rumpe, Martin Schindler and Steven Völkel (2008). 'MontiCore: a framework for the development of textual domain specific languages'. In: *International Conference on Software Engineering (ICSE), Companion Volume*, pages 925–926. DOI: 10.1145/1370175.1370190 (cited on page 3).

Hartman, Toine (Feb. 2022). 'Optimising First-Class Pattern Match Compilation'. Master's thesis. Delft University of Technology. URL: http://resolver.tudelft.nl/uuid:414026ac-b08e-49f3-8aca-1367766161bb (cited on page 11).

Hedin, Görel (2000). 'Reference Attributed Grammars'. In: *Informatica* 24.3, pages 301–317 (cited on page 57).

Hemel, Zef, Danny M. Groenewegen, Lennart C. L. Kats and Eelco Visser (2011). 'Static consistency checking of web applications with WebDSL'. In: *Journal of Symbolic Computation* 46.2, pages 150–182. DOI: 10.1016/j.jsc.2010.08.006 (cited on pages 111–112).

Hemel, Zef, Lennart C. L. Kats, Danny M. Groenewegen and Eelco Visser (2010). 'Code generation by model transformation: a case study in transformation modularity'. In: *Software and Systems Modeling* 9.3, pages 375–402. DOI: 10.1007/s10270-009-0136-1 (cited on page 7).

Herman, David, Aaron Tomb and Cormac Flanagan (2010). 'Space-efficient gradual typing'. In: *Higher-Order and Symbolic Computation* 23.2, pages 167–189. DOI: 10.1007/s10990-011-9066-z (cited on pages 100, 108, 112).

Hills, Mark (2014). 'Streamlining Control Flow Graph Construction with DCFlow'. In: *Software Language Engineering (SLE), Proceedings*. Volume 8706. Lecture Notes in Computer Science, pages 322–341. DOI: 10.1007/978-3-319-11245-9_18 (cited on page 60).

Hoffmann, Christoph Martin and Michael James O'Donnell (Jan. 1982). 'Pattern Matching in Trees'. In: *Journal of the ACM* 29.1, pages 68–95. DOI: 10.1145/322290.322295 (cited on page 128).

Homer, Michael, James Noble, Kim B. Bruce, Andrew P. Black and David J. Pearce (2012). 'Patterns as objects in grace'. In: *Dynamic Languages Symposium (DLS), Proceedings*, pages 17–28. DOI: 10.1145/2384577.2384581 (cited on page 127).

Hong, Sungpack, Hassan Chafi, Eric Sedlar and Kunle Olukotun (2012). 'Green-Marl: a DSL for easy and efficient graph analysis'. In: *Architectural Support for Programming Languages and Operating Systems (ASPLOS), Proceedings*, pages 349–362. DOI: 10.1145/2150976.2151013 (cited on pages 9, 45).

Hong, Sungpack, Martin Sevenich and Jan Lugt (May 2014). *gm_comp, a compiler for Green-Marl written in C++*. URL: https://github.com/stanford-ppl/Green-Marl/tree/4c0d62e67d431d535ca27140df60b25c234a808b (visited on 10th Apr. 2018) (cited on page 16).

Horwitz, Susan, Alan J. Demers and Tim Teitelbaum (1987). 'An Efficient General Iterative Algorithm for Dataflow Analysis'. In: *Acta Informatica* 24.6, pages 679–694 (cited on page 43).

Jay, C. Barry and Delia Kesner (2009). 'First-class patterns'. In: *Journal of Functional Programming* 19.2, pages 191–225. DOI: 10.1017/S0956796808007144 (cited on page 127).

Jetbrains (2018). *Type Checks and Casts: 'is' and 'as' - Kotlin Programming Language*. URL: https://kotlinlang.org/docs/reference/typecasts.html#smart-casts (visited on 12th Apr. 2018) (cited on page 20).

Johann, Patricia and Eelco Visser (2001). 'Fusing Logic and Control with Local Transformations: An Example Optimization'. In: *Electronic Notes in Theoretical Computer Science* 57, pages 144–162. DOI: 10.1016/S1571-0661(04)00271-3 (cited on page 139).

Jørgensen, Jesper (1990). 'Generating a Pattern Matching Compiler by Partial Evaluation'. In: *Glasgow Workshop on Functional Programming, Proceedings*. Workshops in Computing, pages 177–195. DOI: 10.1007/978-1-4471-3810-5_15 (cited on page 128).

Jourdan, Martin and Didier Parigot (1990). 'Techniques for Improving Grammar Flow Analysis'. In: *European Symposium on Programming (ESOP), Proceedings*. Volume 432. Lecture Notes in Computer Science, pages 240–255. DOI: 10.1007/3-540-52592-0_67 (cited on page 43).

Kam, John B. and Jeffrey D. Ullman (1976). 'Global Data Flow Analysis and Iterative Algorithms'. In: *Journal of the ACM* 23.1, pages 158–171. DOI: 10.1145/321921.321938 (cited on page 43).

Kam, John B. and Jeffrey D. Ullman (1977). 'Monotone Data Flow Analysis Frameworks'. In: *Acta Informatica* 7, pages 305–317. DOI: 10.1007/BF00290339 (cited on pages 21, 23, 57).

Kaminski, Ted, Lucas Kramer, Travis Carlson and Eric Van Wyk (2017). 'Reliable and automatic composition of language extensions to C: the ableC extensible language framework'. In: *Proceedings of the ACM on Programming Languages* 1.OOPSLA. DOI: 10.1145/3138224 (cited on page 85).

Karp, Richard M., Raymond E. Miller and Arnold L. Rosenberg (1972). 'Rapid Identification of Repeated Patterns in Strings, Trees and Arrays'. In: *Symposium on Theory of Computing (STOC), Conference Record*, pages 125–136. DOI: 10.1145/800152.804905 (cited on page 127).

Kats, Lennart C. L., Martin Bravenboer and Eelco Visser (2008). 'Mixing source and bytecode: a case for compilation by normalization'. In: *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), Proceedings*, pages 91–108. DOI: 10.1145/1449764.1449772 (cited on page 111).

Kats, Lennart C. L., Karl Trygve Kalleberg and Eelco Visser (2010). 'Domain-Specific Languages for Composable Editor Plugins'. In: *Electronic Notes in Theoretical Computer Science* 253.7, pages 149–163. DOI: 10.1016/j.entcs.2010.08.038 (cited on page 41).

Kats, Lennart C. L., Anthony M. Sloane and Eelco Visser (2009). 'Decorated Attribute Grammars: Attribute Evaluation Meets Strategic Programming'. In: *Compiler Construction (CC), Proceedings*. Volume 5501. Lecture Notes in Computer Science, pages 142–157. DOI: 10.1007/978-3-642-00722-4_11 (cited on page 58).

Kats, Lennart C. L., Rob Vermaas and Eelco Visser (2011). 'Integrated language definition testing: enabling test-driven language development'. In: *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), Proceedings*, pages 139–154. DOI: 10.1145/2048066.2048080 (cited on page 7).

Kats, Lennart C. L. and Eelco Visser (2010). 'The Spoofax language workbench: rules for declarative specification of languages and IDEs'. In: *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), Proceedings*, pages 444–463. DOI: 10.1145/1869459.1869497 (cited on pages 3, 17, 41, 90, 111, 133).

Kieburtz, Richard B., W. Barabash and C. R. Hill (1978). 'A Type-Checking Program Linkage System for Pascal'. In: *International Conference on Software Engineering (ICSE), Proceedings*, pages 23–28 (cited on page 84).

Kildall, Gary Arlen (1973). 'A Unified Approach to Global Program Optimization'. In: *Principles of Programming Languages (POPL), Proceedings*. POPL '73, pages 194–206. DOI: 10.1145/512927.512945 (cited on page 57).

Kiselyov, Oleg (2010). 'Typed Tagless Final Interpreters'. In: *Spring School on Generic and Indexed Programming (SSGIP), Revised Lectures*. Volume 7470. Lecture Notes in Computer Science, pages 130–174. DOI: 10.1007/978-3-642-32202-0_3 (cited on page 85).

Klint, Paul, Tijs van der Storm and Jurgen J. Vinju (2009). 'EASY Metaprogramming with Rascal'. In: *Generative and Transformational Techniques in Software Engineering (GTTSE), Revised Papers*. Volume 6491. Lecture Notes in Computer Science, pages 222–289. DOI: 10.1007/978-3-642-18023-1_6 (cited on pages 3, 60).

Knuth, Donald E. (1968). 'Semantics of Context-Free Languages'. In: *Theory of Computing Systems* 2.2, pages 127–145. DOI: 10.1007/BF01692511 (cited on page 57).

Kogut, Bruce and Anca Metiu (June 2001). 'Open-Source Software Development and Distributed Innovation'. In: *Oxford Review of Economic Policy* 17.2, pages 248–264. DOI: 10.1093/oxrep/17.2.248 (cited on page 2).

Konat, Gabriël, Sebastian Erdweg and Eelco Visser (2018). 'Scalable incremental building with dynamic task dependencies'. In: *Automated Software Engineering (ASE), Proceedings*, pages 76–86. DOI: 10.1145/3238147.3238196 (cited on pages 9, 65, 76, 84).

Konat, Gabriël, Michael J. Steindorfer, Sebastian Erdweg and Eelco Visser (2018). 'PIE: A Domain-Specific Language for Interactive Software Development Pipelines'. In: *The Art, Science, and Engineering of Programming* 2.3, page 9. DOI: 10.22152/programming-journal.org/2018/2/9 (cited on pages 65, 76, 84, 90).

Kosaraju, Sambasiva Rao (1978). 'Strong-connectivity algorithm'. unpublished manuscript (cited on page 46).

Lämmel, Ralf (2003). 'Typed generic traversal with term rewriting strategies'. In: *Journal of Logic and Algebraic Programming* 54.1-2, pages 1–64. DOI: 10.1016/S1567-8326(02)00028-0 (cited on pages 90, 97, 101, 111).

Lämmel, Ralf and Simon L. Peyton Jones (2003). 'Scrap your boilerplate: a practical design pattern for generic programming'. In: *Types in Languages Design and Implementation (TLDI), Proceedings*, pages 26–37. DOI: `10.1145/604174.604179` (cited on pages 90, 111).

Lämmel, Ralf and Joost Visser (2002). 'Typed Combinators for Generic Traversal'. In: *Practical Aspects of Declarative Languages (PADL), Proceedings*. Volume 2257. Lecture Notes in Computer Science, pages 137–154. DOI: `10.1007/3-540-45587-6_10` (cited on pages 90, 111).

Lämmel, Ralf and Joost Visser (2003). 'A Strafunski Application Letter'. In: *Practical Aspects of Declarative Languages (PADL), Proceedings*. Volume 2562. Lecture Notes in Computer Science, pages 357–375. DOI: `10.1007/3-540-36388-2_24` (cited on page 111).

Lippert, Eric (June 2012). *Persistence, façades and Roslyn's red-green trees*. URL: `https://ericlippert.com/2012/06/08/red-green-trees/` (visited on 9th Dec. 2022) (cited on page 138).

Luttik, Bas and Eelco Visser (Nov. 1997). 'Specification of Rewriting Strategies'. In: *Theory and Practice of Algebraic Specifications, Proceedings*. Electronic Workshops in Computing. ISBN: 3540762280 (cited on page 111).

Madsen, Magnus, Ming-Ho Yee and Ondrej Lhoták (2016). 'From Datalog to flix: a declarative language for fixed points on lattices'. In: *Programming Language Design and Implementation (PLDI), Proceedings*, pages 194–208. DOI: `10.1145/2908080.2908096` (cited on page 59).

Magnusson, Eva, Torbjörn Ekman and Gorel Hedin (2007). 'Extending Attribute Grammars with Collection Attributes–Evaluation and Applications'. In: *Source Code Analysis and Manipulation, IEEE International Workshop on*. LU-CS-TR:2012-249 0. DOI: `10.1109/SCAM.2007.13` (cited on page 57).

Magnusson, Eva and Görel Hedin (2007). 'Circular reference attributed grammars - their evaluation and applications'. In: *Science of Computer Programming* 68.1, pages 21–37. DOI: `10.1016/j.scico.2005.06.005` (cited on page 57).

Maranget, Luc (1992). 'Compiling Lazy Pattern Matching'. In: *LISP and Functional Programming (LFP), Proceedings*, pages 21–31. DOI: `10.1145/141471.141499` (cited on page 128).

Maranget, Luc (1994). *Two Techniques for Compiling Lazy Pattern Matching*. Research Report RR-2385. Projet PARA. INRIA. URL: `https://hal.inria.fr/inria-00074292` (cited on pages 116, 128).

Maranget, Luc (2008). 'Compiling pattern matching to good decision trees'. In: *Workshop on ML, Proceedings*. ML'08, pages 35–46. DOI: `10.1145/1411304.1411311` (cited on pages 116, 120, 129).

Meyer, Jon and Troy Downing (1997). *Java Virtual Machine*. ISBN: 1-56592-194-1 (cited on page 109).

Mokhov, Andrey, Neil Mitchell and Simon L. Peyton Jones (July 2018). 'Build systems à la carte'. In: *Proceedings of the ACM on Programming Languages* 2.ICFP. DOI: `10.1145/3236774` (cited on page 84).

Mosses, Peter D. (2004). 'Modular structural operational semantics'. In: *Journal of Logic and Algebraic Programming* 60-61, pages 195–228. DOI: 10.1016/j.jlap.2004.03.008 (cited on page 6).

Muehlboeck, Fabian and Ross Tate (2017). 'Sound gradual typing is nominally alive and well'. In: *Proceedings of the ACM on Programming Languages* 1.OOPSLA. DOI: 10.1145/3133880 (cited on page 112).

Nedjah, Nadia and Luiza de Macedo Mourelle (2001). 'Improving Space, Time, and Termination in Rewriting-Based Programming'. In: *Engineering of Intelligent Systems*. Volume 2070. Lecture Notes in Computer Science, pages 880–890. DOI: 10.1007/3-540-45517-5_97 (cited on page 129).

Nedjah, Nadia, Colin D. Walter and Stephen E. Eldridge (1997). 'Optimal Left-to-Right Pattern-Matching Automata'. In: *Algebraic and Logic Programming (ALP), Proceedings*. Volume 1298. Lecture Notes in Computer Science, pages 273–286. DOI: 10.1007/BFb0027016 (cited on pages 116, 120, 128).

Néron, Pierre, Andrew P. Tolmach, Eelco Visser and Guido Wachsmuth (2015). 'A Theory of Name Resolution'. In: *European Symposium on Programming (ESOP), Proceedings*. Volume 9032. Lecture Notes in Computer Science, pages 205–231. DOI: 10.1007/978-3-662-46669-8_9 (cited on pages 6, 41).

New, Max S., Daniel R. Licata and Amal Ahmed (2021). 'Gradual type theory'. In: *Journal of Functional Programming* 31. DOI: 10.1017/S0956796821000125 (cited on page 138).

Nielsen, Jakob (1993). *Usability engineering*. ISBN: 978-0-12-518405-2 (cited on page 1).

Nielson, Flemming, Hanne Riis Nielson and Chris Hankin (2005). *Principles of program analysis (2. corr. print)*. DOI: 10.1007/978-3-662-03811-6 (cited on pages 16, 21, 24, 45).

Odersky, Martin and M. Zenger (2005). 'Independently extensible solutions to the expression problem'. In: *Foundations of Object-Oriented Languages (FOOL), Proceedings* (cited on page 85).

Olmos, Karina and Eelco Visser (2005). 'Composing Source-to-Source Data-Flow Transformations with Rewriting Strategies and Dependent Dynamic Rewrite Rules'. In: *Compiler Construction (CC), Proceedings*. Volume 3443. Lecture Notes in Computer Science, pages 204–220. DOI: 10.1007/978-3-540-31985-6_14 (cited on pages 7, 112).

Oracle Corporation (2015a). *PGX 1.1.0 Documentation – Closeness Centrality*. URL: https://docs.oracle.com/cd/E56133_01/1.1.0/reference/algorithms/closeness_centrality.html (visited on 27th Feb. 2018) (cited on page 46).

Oracle Corporation (2015b). *PGX 1.1.0 Documentation – List of Built-in Algorithms*. URL: https://docs.oracle.com/cd/E56133_01/1.1.0/reference/algorithms/index.html (visited on 27th Feb. 2018) (cited on page 50).

Overmars, Mark H and Jan van Leeuwen (1979). 'Rapid subtree identification revisited'. In: Technical report. URL: http://www.cs.uu.nl/research/techreps/repo/CS-1979/1979-03.pdf (cited on page 128).

Parr, Terence John and Russell W. Quong (1995). 'ANTLR: A Predicated-LL(k) Parser Generator'. In: *Software: Practice and Experience* 25.7, pages 789–810. DOI: 10.1002/spe.4380250705 (cited on page 138).

Pearce, David J. and James Noble (July 2011). *Structural and Flow-sensitive types for Whiley*. Technical report. School of Engineering and Computer Science, Victoria University of Wellington. URL: https://ecs.wgtn.ac.nz/foswiki/pub/Main/TechnicalReportSeries/ECSTR10-23.pdf (cited on page 20).

Pech, Vaclav (2021). 'JetBrains MPS: Why Modern Language Workbenches Matter'. In: *Domain-Specific Languages in Practice: with JetBrains MPS*, pages 1–22. DOI: 10.1007/978-3-030-73758-0_1 (cited on page 3).

Petrashko, Dmitry, Ondrej Lhoták and Martin Odersky (2017). 'Miniphases: compilation using modular and efficient tree transformations'. In: *Programming Language Design and Implementation (PLDI), Proceedings*, pages 201–216. DOI: 10.1145/3062341.3062346 (cited on page 139).

Pettersson, Mikael (1992). 'A Term Pattern-Match Compiler Inspired by Finite Automata Theory'. In: *Compiler Construction (CC), Proceedings*. Volume 641. Lecture Notes in Computer Science, pages 258–270. DOI: 10.1007/3-540-55984-1_24 (cited on page 128).

Pettersson, Mikael (1999). 'Compiling Pattern Matching'. In: *Compiling Natural Semantics*. Volume 1549. Lecture Notes in Computer Science. Chapter 7, pages 85–109. DOI: 10.1007/10693148_7 (cited on page 129).

Red Hat, Inc. (Apr. 2018). *Eclipse Ceylon: Quick Introduction*. URL: https://ceylon-lang.org/documentation/1.3/introduction/#typesafe_null_and_flow_sensitive_typing (visited on 12th Apr. 2018) (cited on page 20).

Reiss, Steven P. (1984). 'An approach to incremental compilation'. In: *Symposium on Compiler Construction, Proceedings*, pages 144–156. DOI: 10.1145/502874.502889 (cited on pages 64, 84).

Reynaud, Daniel (Mar. 2006). *JasminXT Syntax*. URL: http://jasmin.sourceforge.net/xt.html (visited on 13th Mar. 2018) (cited on page 109).

Reynolds, John C. (1983). 'Types, Abstraction and Parametric Polymorphism'. In: *World Computer Congress (WCC), Proceedings*, pages 513–523 (cited on page 98).

Rountev, Atanas, Barbara G. Ryder and William Landi (1999). 'Data-Flow Analysis of Program Fragments'. In: *European Software Engineering Conference / Foundations of Software Engineering (ESEC/FSE), Jointly Held Proceedings*. Volume 1687. Lecture Notes in Computer Science, pages 235–252. DOI: 10.1007/3-540-48166-4_15 (cited on page 137).

Runeson, Per and Martin Höst (2009). 'Guidelines for conducting and reporting case study research in software engineering'. In: *Empirical Software Engineering* 14.2, pages 131–164. DOI: 10.1007/s10664-008-9102-8 (cited on page 4).

Rust Project Developers (Feb. 2018). *librustc – builtin lints*. URL: https://github.com/rust-lang/rust/blob/67712d79451b042b6fca2167c1a57d91d86f663b/src/librustc/lint/builtin.rs#L76-L80 (visited on 10th Apr. 2018) (cited on page 16).

Ryan, James L., Richard L. Crandall and Marion C. Medwedeff (1966). 'A conversational system for incremental compilation and execution in a time-sharing environment'. In: *American Federation of Information Processing Societies (AFIPS), Proceedings*. Volume 29, pages 1–21. DOI: 10.1145/1464291.1464293 (cited on pages 64, 84).

Sakka, Laith, Kirshanthan Sundararajah and Milind Kulkarni (2017). 'TreeFuser: a framework for analyzing and fusing general recursive tree traversals'. In: *Proceedings of the ACM on Programming Languages* 1.OOPSLA. DOI: 10.1145/3133900 (cited on page 139).

Schnoebelen, Philippe (1988). 'Refined Compilation of Pattern-Matching for Functional Languages'. In: *Science of Computer Programming* 11.2, pages 133–159. DOI: 10.1016/0167-6423(88)90002-0 (cited on page 120).

Schorre, D. Val (1964). 'META II a syntax-oriented compiler writing language'. In: *ACM National Conference, Proceedings*, page 41. DOI: 10.1145/800257.808896 (cited on page 1).

Schubert, Philipp Dominik, Ben Hermann and Eric Bodden (2021). 'Lossless, Persisted Summarization of Static Callgraph, Points-To and Data-Flow Analysis'. In: *European Conference on Object-Oriented Programming (ECOOP), Proceedings*. Volume 194. LIPIcs. DOI: 10.4230/LIPIcs.ECOOP.2021.2 (cited on page 137).

Scott, Kevin and Norman Ramsey (Mar. 1999). *When Do Match-Compilation Heuristics Matter?* Technical Report. University of Virginia Dept. of Computer Science. DOI: 10.18130/V3GB4M (cited on page 129).

Sekar, R. C., Ramashubramani Ramesh and I. V. Ramakrishnan (1995). 'Adaptive Pattern Matching'. In: *SIAM Journal on Computing* 24.6, pages 1207–1234. DOI: 10.1137/S0097539793246252 (cited on pages 120, 129).

Sestoft, Peter (1996). 'ML Pattern Match Compilation and Partial Evaluation'. In: *Partial Evaluation*. Volume 1110. Lecture Notes in Computer Science, pages 446–464. DOI: 10.1007/3-540-61580-6_22 (cited on page 128).

Shivers, Olin (1988). 'Higher-order control-flow analysis in retrospect: lessons learned, lessons abandoned (with retrospective)'. In: *20 Years of the ACM SIGPLAN Conference on Programming Language Design and Implementation 1979-1999, A Selection*, pages 257–269. DOI: 10.1145/989393.989421 (cited on page 20).

Siek, Jeremy G., Ronald Garcia and Walid Taha (2009). 'Exploring the Design Space of Higher-Order Casts'. In: *European Symposium on Programming (ESOP), Proceedings*. Volume 5502. Lecture Notes in Computer Science, pages 17–31. DOI: 10.1007/978-3-642-00590-9_2 (cited on page 101).

Siek, Jeremy G. and Walid Taha (Sept. 2006). 'Gradual typing for functional languages'. In: *Scheme and Functional Programming Workshop*. Volume 6, pages 81–92 (cited on pages 10, 90, 99, 112).

Siek, Jeremy G. and Walid Taha (2007). 'Gradual Typing for Objects'. In: *European Conference on Object-Oriented Programming (ECOOP), Proceedings*. Volume 4609. Lecture Notes in Computer Science, pages 2–27. DOI: 10.1007/978-3-540-73589-2_2 (cited on page 90).

Sloane, Anthony M., Matthew Roberts and Leonard G. C. Hamey (2014). 'Respect Your Parents: How Attribution and Rewriting Can Get Along'. In: *Software Language Engineering (SLE), Proceedings*. Volume 8706. Lecture Notes in Computer Science, pages 191–210. DOI: 10.1007/978-3-319-11245-9_11 (cited on page 58).

Smaragdakis, Yannis and George Balatsouras (2015). 'Pointer Analysis'. In: *Foundations and Trends in Programming Languages* 2.1, pages 1–69. DOI: 10.1561/2500000014 (cited on page 59).

Smaragdakis, Yannis, Martin Bravenboer and Ondrej Lhoták (2011). 'Pick your contexts well: understanding object-sensitivity'. In: *Principles of Programming Languages (POPL), Proceedings*, pages 17–30. DOI: 10.1145/1926385.1926390 (cited on page 21).

Smits, Jeff, Toine Hartman and Jesper Cockx (2022). 'Optimising First-Class Pattern Matching'. In: *Software Language Engineering (SLE), Proceedings*, pages 74–83. DOI: 10.1145/3567512.3567519 (cited on pages 12, 115, 189).

Smits, Jeff, Gabriël Konat and Eelco Visser (2020). 'Constructing Hybrid Incremental Compilers for Cross-Module Extensibility with an Internal Build System'. In: *The Art, Science, and Engineering of Programming* 4.3, 16. DOI: 10.22152/programming-journal.org/2020/4/16 (cited on pages 12, 63, 127, 189).

Smits, Jeff and Eelco Visser (2017). 'FlowSpec: declarative dataflow analysis specification'. In: *Software Language Engineering (SLE), Proceedings*, pages 221–231. DOI: 10.1145/3136014.3136029 (cited on pages 18, 189).

Smits, Jeff and Eelco Visser (2020). 'Gradually typing strategies'. In: *Software Language Engineering (SLE), Proceedings*, pages 1–15. DOI: 10.1145/3426425.3426928 (cited on pages 12, 89, 189).

Smits, Jeff, Guido Wachsmuth and Eelco Visser (2020). 'FlowSpec: A Declarative Specification Language for Intra-Procedural Flow-Sensitive Data-Flow Analysis'. In: *Journal of Computer Languages* 57, 100924, page 39. DOI: 10.1016/j.cola.2019.100924 (cited on pages 12, 15, 189).

Söderberg, Emma, Torbjörn Ekman, Görel Hedin and Eva Magnusson (2013). 'Extensible intraprocedural flow analysis at the abstract syntax tree level'. In: *Science of Computer Programming* 78.10, pages 1809–1827. DOI: 10.1016/j.scico.2012.02.002 (cited on page 57).

Söderberg, Emma and Görel Hedin (2012). *Incremental Evaluation of Reference Attribute Grammars using Dynamic Dependency Tracking*. Technical report 98. Department of Computer Science, Lund University. URL: https://portal.research.lu.se/files/3312049/2543179.pdf (cited on page 84).

Szabó, Tamás, Simon Alperovich, Markus Völter and Sebastian Erdweg (2016). 'An extensible framework for variable-precision data-flow analyses in MPS'. In: *Automated Software Engineering (ASE), Proceedings*, pages 870–875. DOI: 10.1145/2970276.2970296 (cited on page 59).

Szabó, Tamás, Sebastian Erdweg and Markus Völter (2016). 'IncA: a DSL for the definition of incremental program analyses'. In: *Automated Software Engineering (ASE), Proceedings*, pages 320–331. DOI: 10.1145/2970276.2970298 (cited on page 60).

Szabó, Tamás, Markus Völter and Sebastian Erdweg (2017). *IncA$_L$: A DSL for Incremental Program Analysis with Lattices*. Talk proposal; full article not published at the time (cited on page 60).

Tarjan, Robert Endre (1972). 'Depth-First Search and Linear Graph Algorithms'. In: *SIAM Journal on Computing* 1.2, pages 146–160 (cited on pages 43, 159–160).

TIOBE (Nov. 2022). *TIOBE Index for November 2022*. URL: `https://www.tiobe.com/tiobe-index/` (visited on 23rd Nov. 2022) (cited on page 2).

Torgersen, Mads (2004). 'The Expression Problem Revisited'. In: *European Conference on Object-Oriented Programming (ECOOP), Proceedings*. Volume 3086. Lecture Notes in Computer Science, pages 123–143. DOI: `10.1007/978-3-540-24851-4_6` (cited on page 85).

Tullsen, Mark (2000). 'First Class Patterns'. In: *Practical Aspects of Declarative Languages (PADL), Proceedings*. Volume 1753. Lecture Notes in Computer Science, pages 1–15. DOI: `10.1007/3-540-46584-7_1` (cited on page 127).

Turner, David (1983). *SASL Language Manual* (cited on page 127).

Van den Brand, Mark G. J., H. A. de Jong, Paul Klint and Pieter A. Olivier (2000). 'Efficient annotated terms'. In: *Software: Practice and Experience* 30.3, pages 259–291. DOI: `10.1002/(SICI)1097-024X(200003)30:3%3C259::AID-SPE298%3E3.0.CO;2-Y` (cited on page 116).

Van den Brand, Mark G. J., 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 (2001). 'The ASF+SDF Meta-environment: A Component-Based Language Development Environment'. In: *Compiler Construction (CC), Proceedings*. Volume 2027. Lecture Notes in Computer Science, pages 365–370. DOI: `10.1016/S1571-0661(04)80917-4` (cited on page 3).

Van Wyk, Eric (17th June 2019). personal correspondence (cited on page 86).

Van Wyk, Eric, Derek Bodin, Jimin Gao and Lijesh Krishnan (2010). 'Silver: An extensible attribute grammar system'. In: *Science of Computer Programming* 75.1-2, pages 39–54. DOI: `10.1016/j.scico.2009.07.004` (cited on pages 3, 58, 85).

Van Antwerpen, Hendrik, Pierre Néron, Andrew P. Tolmach, Eelco Visser and Guido Wachsmuth (2016). 'A constraint language for static semantic analysis based on scope graphs'. In: *Partial Evaluation and Program Manipulation (PEPM), Proceedings*, pages 49–60. DOI: `10.1145/2847538.2847543` (cited on pages 17, 41).

Van Binsbergen, L. Thomas, Mauricio Verano Merino, Pierre Jeanjean, Tijs van der Storm, Benoît Combemale and Olivier Barais (2020). 'A principled approach to REPL interpreters'. In: *New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!), Proceedings*, pages 84–100. DOI: `10.1145/3426428.3426917` (cited on page 139).

Van Deursen, Arie, Paul Klint and Joost Visser (2000). 'Domain-Specific Languages: An Annotated Bibliography'. In: *SIGPLAN Notices* 35.6, pages 26–36. DOI: `10.1145/352029.352035` (cited on page 2).

Vergu, Vlad A., Pierre Néron and Eelco Visser (2015). 'DynSem: A DSL for Dynamic Semantics Specification'. In: *Rewriting Techniques and Applications (RTA), Proceedings*. Volume 36. LIPIcs, pages 365–378. DOI: `10.4230/LIPIcs.RTA.2015.365` (cited on page 6).

Visser, Eelco (1999). 'Strategic Pattern Matching'. In: *Rewriting Techniques and Applications (RTA), Proceedings*. Volume 1631. Lecture Notes in Computer Science, pages 30–44. DOI: `10.1007/3-540-48685-2_3` (cited on pages 69, 98, 111, 127).

Visser, Eelco (2001a). 'Scoped Dynamic Rewrite Rules'. In: *Electronic Notes in Theoretical Computer Science* 59.4, pages 375–396. DOI: `10.1016/S1571-0661(04)00298-1` (cited on page 112).

Visser, Eelco (2001b). 'Stratego: A Language for Program Transformation Based on Rewriting Strategies'. In: *Rewriting Techniques and Applications (RTA), Proceedings*. Volume 2051. Lecture Notes in Computer Science, pages 357–362. DOI: `10.1007/3-540-45127-7_27` (cited on pages 116, 127).

Visser, Eelco (Oct. 2005). 'Transformations for Abstractions'. In: *Source Code Analysis and Manipulation (SCAM), Proceedings*. DOI: `10.1109/SCAM.2005.26` (cited on page 66).

Visser, Eelco (2007). 'WebDSL: A Case Study in Domain-Specific Language Engineering'. In: *Generative and Transformational Techniques in Software Engineering (GTTSE), Revised Papers*. Volume 5235. Lecture Notes in Computer Science, pages 291–373. DOI: `10.1007/978-3-540-88643-3_7` (cited on page 69).

Visser, Eelco (Jan. 2013). *strategoxt/bound-unbound-vars*. URL: `https://github.com/metaborg/strategoxt/blob/76689003f94bcd51c84712bf4509b706dd34d9ab/strategoxt/stratego-libraries/strc/lib/stratego/strc/opt/bound-unbound-vars.str` (visited on 1st Nov. 2022) (cited on page 174).

Visser, Eelco (Feb. 2021). *A Brief History of the Spoofax Language Workbench*. URL: `https://eelcovisser.org/blog/2021/02/08/spoofax-mip/` (visited on 9th Nov. 2022) (cited on page 4).

Visser, Eelco and Zine-El-Abidine Benaissa (1998). 'A core language for rewriting'. In: *Electronic Notes in Theoretical Computer Science* 15, pages 422–441. DOI: `10.1016/S1571-0661(05)80027-1` (cited on pages 101–102, 111, 116–118, 127).

Visser, Eelco, Zine-El-Abidine Benaissa and Andrew P. Tolmach (1998). 'Building Program Optimizers with Rewriting Strategies'. In: *International Conference on Functional Programming (ICFP), Proceedings*, pages 13–26. DOI: `10.1145/289423.289425` (cited on pages 8, 90–91, 93, 111, 116, 127).

Visser, Eelco, Guido Wachsmuth, Andrew P. Tolmach, Pierre Néron, Vlad A. Vergu, Augusto Passalaqua and Gabriël Konat (2014). 'A Language Designer's Workbench: A One-Stop-Shop for Implementation and Verification of Language Designs'. In: *New Ideas, New Paradigms, and Reflections on Programming & Software (Onward!), Proceedings*, pages 95–111. DOI: `10.1145/2661136.2661149` (cited on page 5).

Vogt, Harald, S. Doaitse Swierstra and Matthijs F. Kuiper (1989). 'Higher-Order Attribute Grammars'. In: *Programming Language Design and Implementation (PLDI), Proceedings*, pages 131–145. DOI: 10.1145/73141.74830 (cited on page 57).

Vollebregt, Tobi, Lennart C. L. Kats and Eelco Visser (2012). 'Declarative specification of template-based textual editors'. In: *Language Descriptions, Tools, and Applications (LDTA), Proceedings*, pages 1–7. DOI: 10.1145/2427048.2427056 (cited on page 17).

Wadler, Philip (1989). 'Theorems for Free!' In: *FPCA*, pages 347–359. DOI: 10.1145/99370.99404 (cited on page 98).

Wadler, Philip (Nov. 1998). *The expression problem*. Java-genericity mailing list. URL: https://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt (visited on 16th Dec. 2019) (cited on pages 65, 85).

Wang, Yanlin and Bruno C. D. S. Oliveira (2016). 'The expression problem, trivially!' In: *International Conference on Modularity, Proceedings*, pages 37–41. DOI: 10.1145/2889443.2889448 (cited on page 85).

Wimmer, Christian and Thomas Würthinger (2012). 'Truffle: a self-optimizing runtime system'. In: *Systems, Programming, and Applications: Software for Humanity (SPLASH), Proceedings*, pages 13–14. DOI: 10.1145/2384716.2384723 (cited on page 42).

Wirth, Niklaus (2007). 'Modula-2 and Oberon'. In: *History of Programming Languages Conference (HOPL), Proceedings*, pages 1–10. DOI: 10.1145/1238844.1238847 (cited on page 84).

Woerister, Michael (8th Sept. 2016). *Incremental Compilation*. URL: https://blog.rust-lang.org/2016/09/08/incremental.html (visited on 12th Dec. 2019) (cited on page 85).

Woerister, Michael (29th Jan. 2019). *Tracking Issue for making incremental compilation the default for Release Builds*. URL: https://github.com/rust-lang/rust/issues/57968 (visited on 12th Dec. 2019) (cited on page 85).

Xie, Ningning, Xuan Bi and Bruno C. D. S. Oliveira (2018). 'Consistent Subtyping for All'. In: *European Symposium on Programming (ESOP), Proceedings*. Volume 10801. Lecture Notes in Computer Science, pages 3–30. DOI: 10.1007/978-3-319-89884-1_1 (cited on page 112).

Yang, Xuejun, Yang Chen, Eric Eide and John Regehr (2011). 'Finding and understanding bugs in C compilers'. In: *Programming Language Design and Implementation (PLDI), Proceedings*, pages 283–294. DOI: 10.1145/1993498.1993532 (cited on page 1).

Appendix A

Adapted SCC Algorithm and FlowSpec Case Study Listings

This appendix contains supplementary material of Chapter 2. This includes the full pseudocode of the adapted Strongly Connected Component (SCC) algorithm originally by Tarjan (1972), and the full FlowSpec source code of the GreenMarl and Stratego case studies. Since the listings of this appendix are rather long, we provide a short table of contents here.

Table of Contents

A.1 Strongly Connected Components

The computation of the ordering uses a slightly adapted version of Tarjan's strongly connected components (SCCs) algorithm (Tarjan 1972) in Listing A.1. Tarjan's algorithm already gives the strongly connected components in *reverse* topological order. To get the topological order out, we use a stack instead of an array to add the SCCs to when they are discovered. We also keep an extra stack where nodes of an SCC are added in *postorder*, in contrast to the set or boolean flag which is added in *preorder*. Since the algorithm already does a depth-first search, this gives us a reverse (because of the stack) postorder over the depth-first spanning forest of the SCC.

```
index := 0
S := empty stack
Q := empty stack
for $\ell \in E$: // from the extremal labels of the CFG
  if $\ell$.index is undefined:
    strongconnect($\ell$)

func strongconnect($\ell$):
  $\ell$.index := index
  $\ell$.lowlink := index
  index := index + 1
  $\ell$.onStack := true
  // Note we don't add $\ell$ to the SCC stack here, but the onStack marker is still
   there so the algorithm works unchanged for the next loop

  for ($\ell$, $\ell'$) $\in F$:
    if $\ell'$.index is undefined:
      strongconnect($\ell'$)
      $\ell$.lowlink := min($\ell$.lowlink, $\ell'$.lowlink)
    else if $\ell'$.onStack:
      $\ell$.lowlink := min($\ell$.lowlink, $\ell'$.index)

  // Note that we add $\ell$ to the SCC stack here instead, in *postorder*
  S.push($\ell$)

  if $\ell$.lowlink $=$ $\ell$.index:
    scc := empty array
    do:
      $\ell'$ := S.pop()
      $\ell'$.onStack := false
      scc.push($\ell'$)
    while($\ell' \neq \ell$)
    Q.push(scc)

// output: Q, the stack of SCCs
```

Listing A.1 The adapted version of Tarjan's strongly connected components, which gives topologically ordered strongly connected components (SCCs) where the SCCs have reverse postorder in their depth-first spanning tree.

A.2 GreenMarl Case Study

The following listings were used in the GreenMarl case study. They include the control-flow specification of GreenMarl in FlowSpec, as well as analyses (1) Live Variables, (2) Reaching Definitions, (3) Available Expressions, (4) Very Busy Expressions, and (5) Constant Propagation.

A.2.1 Control-Flow Specification for GreenMarl

The control-flow graph rules are roughly ordered by the corresponding SDF3 files that define the abstract syntax that we match. The file starts with a module definition and the import of the external signature definitions. Then we define general rules for `Cons` and `Nil`, control-flow in lists assume that each element of the list can control-flow and threads the control-flow through the list from left to right. The root of a control-flow graph in GreenMarl is at the procedure level. Blocks have lists of statements, so their control-flow is that of the list. In all kinds of assignments, such as the reduce assignment and the arg-min assignment, the right-hand side expressions are executed before the left-hand side.

```
module control-data

imports
  external
    signatures/-
    signatures/preprocess/Extra-Constructors-sig
    signatures/post-analysis/-
    signatures/frontend/syntax/core/-

control-flow rules // Library rules?

  Cons(head, tail) =
    entry -> head -> tail -> exit

  Nil() = entry -> exit

control-flow rules // gm_lang

  root Proc(_, params, _, _, body) =
    start -> params -> body -> end

control-flow rules // Statements

  Block(statements) =
    entry -> statements -> exit

  ReduceAssign(lhs, _, rhs, _) =
    entry -> rhs -> lhs -> exit

  ArgMinMax(lhs, lhss, _, rhs, rhss, _) =
    entry -> rhs -> rhss -> lhs -> lhss -> exit
```

Listing A.2 The Control-Flow Graph Rules for GreenMarl (1/4)

Loops and traversals show up as loops in the control-flow graph as well. The bounds stand in for the decision to go into the body or go on with the program that follows the loop/traversal. Rules like the one for `NoInReverse()` do not contribute nodes to the control-flow graph. Procedure calls execute their expressions, then do the call itself by using the matched AST node as a control-flow graph node with the this keyword. Printing has side-effects and is therefore also itself put in the control-flow graph. Returns use the end keyword instead of the local exit to connect to the end of the procedure enclosing (since it declared itself a root). When an AST node is directly a control-flow graph node and has no further control-flow inside, we can use the shortcut rule `node` followed by the AST pattern.

```
  ForEach(_, bounds, statement) =
    entry -> bounds -> exit,
             bounds -> statement -> bounds

  BFS(bounds, statement, rev) =
    entry -> bounds -> exit,
             bounds -> statement -> bounds,
             bounds -> rev -> bounds

  DFS(bounds, statement, post) =
    entry -> bounds -> exit,
             bounds -> statement -> bounds,
             bounds -> post -> bounds

  NoInReverse() = entry -> exit

  InReverse(filter, statement) =
    entry -> filter -> statement -> exit

  NoInPost() = entry -> exit

  InPost(filter, statement) =
    entry -> filter -> statement -> exit

  CallStm(call) = entry -> call -> exit

  ProcCallStm(_, exprs, outargs) =
    entry -> exprs -> this -> outargs -> exit

  Print(_, exprs) = entry -> exprs -> this -> exit

  Error(expr) = entry -> expr -> this -> end

  ReturnWith(_, expr) = entry -> expr -> end

  Return() = entry -> end

control-flow rules // Declarations (post-analysis)

  node Decl(_, _, _)
```

Listing A.3 The Control-Flow Graph Rules for GreenMarl (2/4)

```
control-flow rules // Statements (core)

  IfThenElse(cond, thenb, elseb) =
    entry -> cond -> thenb -> exit,
             cond -> elseb -> exit

  While(cond, body) =
    entry -> cond -> exit,
    cond -> body -> cond

  DoWhile(body, cond) =
    entry -> body -> cond -> exit,
    cond -> body

  Assign(lhs, rhs) =
    entry -> rhs -> lhs -> exit

  node VarAssign(_)

  PropAssign(r,_) = entry -> r -> this -> exit

  ElementAssignWrapper(ea, _) =
    entry -> ea -> exit

  ElementAssign(ea,expr) =
    entry -> expr -> ea -> exit

control-flow rules // Iterators

  IterBounds(_, range, order) =
    entry -> range -> order -> this -> exit

  NoOrder() = entry -> exit

  OrderBy(expr, _) =
    entry -> expr -> exit

  node GraphIterRange(_,_)
  node NodeIterRange(_,_)
  node CollectionIterRange(_,_)
  node MapIterRange(_,_)

  VectorIterRange(e) = entry -> e -> this -> exit

  XFSIterBounds(_, range, nav) =
    entry -> range -> nav -> this -> exit

  node BFSRange(_,_,_)
  node DFSRange(_,_,_)

  NoNavigator() = entry -> exit

  Navigator(expr) =
    entry -> expr -> exit
```

Listing A.4 The Control-Flow Graph Rules for GreenMarl (3/4)

```
control-flow rules // Expressions

  node IntLit(_)
  node LongLit(_)
  node FloatLit(_)
  node DoubleLit(_)
  node StringLit(_)
  node NIL()
  node Inf(_, _)
  node True()
  node False()

  Abs(e) = entry -> e -> this -> exit

  node VarRef(_)
  Placeholder() = entry -> exit
  node PropRef(_, _)
  ElementAccess(ea, e) = entry -> e -> ea -> exit

  UMin(e) = entry -> e -> this -> exit
  Mul(e1,e2) = entry -> e1 -> e2 -> this -> exit
  Div(e1,e2) = entry -> e1 -> e2 -> this -> exit
  Mod(e1,e2) = entry -> e1 -> e2 -> this -> exit
  Add(e1,e2) = entry -> e1 -> e2 -> this -> exit
  Sub(e1,e2) = entry -> e1 -> e2 -> this -> exit
  Not(e) = entry -> e -> this -> exit
  And(e1,e2) = entry -> e1 -> e2 -> this -> exit
  Or(e1,e2) = entry -> e1 -> e2 -> this -> exit
  Eq(e1,e2) = entry -> e1 -> e2 -> this -> exit
  Gt(e1,e2) = entry -> e1 -> e2 -> this -> exit
  Lt(e1,e2) = entry -> e1 -> e2 -> this -> exit
  Geq(e1,e2) = entry -> e1 -> e2 -> this -> exit
  Leq(e1,e2) = entry -> e1 -> e2 -> this -> exit
  Neq(e1,e2) = entry -> e1 -> e2 -> this -> exit

  Cast(_,e) = entry -> e -> this -> exit
  TerIf(e1,e2,e3) = entry -> e1 -> e2 -> exit,
                             e1 -> e3 -> exit

control-flow rules // Common

  FuncCall(e1, _, e2) =
    entry -> e1 -> e2 -> this -> exit
  ProcCall(_, e, _) =
    entry -> e -> this -> exit

  NoOutArgs() = entry -> exit
  OutArgs(outargs) = entry -> outargs -> exit
  Ignore() = entry -> exit
```

Listing A.5 The Control-Flow Graph Rules for GreenMarl (4/4)

A.2.2 Live Variables Analysis Specification for GreenMarl

In the figure is a definition of a live variables that tells you which variables may be read before being re-assigned. The Set contains the term that is the name string from the AST. At any assignment the name is removed. At a reading point the name is added. Information is propagated backwards so that you can look into the future of the program when reading the analysis results.

```
properties

  // Live Variables
  live: MaySet(term)

property rules

  live(VarAssign(n) -> next) =
    live(next) \ { Var{n} }

  live(PropAssign(_,p) -> next) =
    live(next) \ { Prop{p} }

  live(IterBounds(n, _, _) -> next) =
    live(next) \ { Var{n} }

  live(XFSIterBounds(n, _, _) -> next) =
    live(next) \ { Var{n} }

  live(this@PropRef(_,p) -> next) =
    live(next) \/ { Prop{p} }

  live(this@VarRef(n) -> next) =
    live(next) \/ { Var{n} }

  live(_ -> next) =
    live(next)
```

Listing A.6 A Live Variables Analysis for GreenMarl

A.2.3 Reaching Definitions Analysis Specification for GreenMarl

Reaching Definitions is similar to the previous analysis but records writes to a variable and passes these forwards. Therefore you can use this analysis at a point where a variable is read to see where the value read there may have originated from. Because local variable declarations are given a 'write' of `None`, you can use this information to track down places where a variable may be uninitialised as well. As a separate analysis this is usually known as Definite Assignment analysis.

```
properties
  reaching: MaySet(term * Option(index))

property rules

  reaching(prev -> this@Decl(n, _, InArg())) =
    { (n, Some(indexOf(this))) } \/ reaching(prev)

  reaching(prev -> this@Decl(n, _, OutArg())) =
    { (n, Some(indexOf(this))) } \/ reaching(prev)

  reaching(prev -> this@Decl(n, _, Local())) =
    { (n, None()) } \/ reaching(prev)

  reaching(prev -> this@VarAssign(n)) =
    { (n, Some(indexOf(this))) } \/
    { (m, l) | (m, l) <- reaching(prev), m != n }

  reaching(prev -> this@PropAssign(_,p)) =
    { (p, Some(indexOf(this))) } \/
    { (m, l) | (m, l) <- reaching(prev), m != p }

  reaching(prev -> this@IterBounds(n, _, _)) =
    { (n, Some(indexOf(this))) } \/
    { (m, l) | (m, l) <- reaching(prev), m != n }

  reaching(prev -> this@XFSIterBounds(n, _, _)) =
    { (n, Some(indexOf(this))) } \/
    { (m, l) | (m, l) <- reaching(prev), m != n }

  // note that we model output effects like printing as writing to an artificial
  //  variable, which allows reasoning about output dependences
  reaching(prev -> this@Print(_,_)) =
    { (Print(), Some(indexOf(this))) } \/
    { (m, l) |
      (m, l) <- reaching(prev),
      m != Print() }

  reaching(prev -> _) =
    reaching(prev)
```

Listing A.7 A Reaching Definitions Analysis for GreenMarl

A.2.4 Available Expressions Analysis Specification for GreenMarl

Available Expressions analysis uses an external property from NaBL2 to collect references from an expression. This is used to filter any expressions that use a name from the set when that name is assigned. We do not need to worry about the scope of names as that is handled by NaBL2.

```
properties
  available: MustSet(term)
  external refs: Set(name)

property rules
  available(prev -> _) = available(prev)

  available(prev -> VarAssign(n)) =
    { expr |
      expr <- available(prev),
      !(Var{n} in refs(expr)) }

  available(prev -> PropAssign(_,p)) =
    { expr |
      expr <- available(prev),
      !(Prop{p} in refs(expr)) }

  available(prev -> this@IterBounds(n, _, _)) =
    { expr |
      expr <- available(prev),
      !(Var{n} in refs(expr)) }

  available(prev -> this@XFSIterBounds(n, _, _)) =
    { expr |
      expr <- available(prev),
      !(Var{n} in refs(expr)) }

  available(prev -> this@Abs(_)) =          available(prev) \/ { this }
  available(prev -> this@UMin(_)) =         available(prev) \/ { this }
  available(prev -> this@Mul(_,_)) =        available(prev) \/ { this }
  available(prev -> this@Div(_,_)) =        available(prev) \/ { this }
  available(prev -> this@Mod(_,_)) =        available(prev) \/ { this }
  available(prev -> this@Add(_,_)) =        available(prev) \/ { this }
  available(prev -> this@Sub(_,_)) =        available(prev) \/ { this }
  available(prev -> this@Not(_)) =         available(prev) \/ { this }
  available(prev -> this@Or(_,_)) =         available(prev) \/ { this }
  available(prev -> this@Eq(_,_)) =         available(prev) \/ { this }
  available(prev -> this@Gt(_,_)) =         available(prev) \/ { this }
  available(prev -> this@Lt(_,_)) =         available(prev) \/ { this }
  available(prev -> this@Geq(_,_)) =        available(prev) \/ { this }
  available(prev -> this@Leq(_,_)) =        available(prev) \/ { this }
  available(prev -> this@Neq(_,_)) =        available(prev) \/ { this }
  available(prev -> this@Cast(_,_)) =       available(prev) \/ { this }
  available(prev -> this@TerIf(_,_,_)) =    available(prev) \/ { this }
  available(prev -> this@FuncCall(_,_,_)) = available(prev) \/ { this }
  available(prev -> this@ProcCall(_,_,_)) = available(prev) \/ { this }
```

Listing A.8 Available Expressions Analysis for GreenMarl

A.2.5 Very Busy Expressions Analysis Specification for GreenMarl

Very Busy Expressions analysis is very similar to Available Expressions analysis, except is goes backwards.

```
properties
  veryBusy: MustSet(term)

property rules
  veryBusy(_ -> next) = veryBusy(next)

  veryBusy(VarAssign(n) -> next) =
    { expr |
      expr <- veryBusy(next),
      !(Var{n} in refs(expr)) }

  veryBusy(PropAssign(_,p) -> next) =
    { expr |
      expr <- veryBusy(next),
      !(Prop{p} in refs(expr)) }

  veryBusy(this@IterBounds(n, _, _) -> next) =
    { expr |
      expr <- veryBusy(next),
      !(Var{n} in refs(expr)) }

  veryBusy(this@XFSIterBounds(n, _, _) -> next) =
    { expr |
      expr <- veryBusy(next),
      !(Var{n} in refs(expr)) }

  veryBusy(this@Abs(_) -> next) =          veryBusy(next) \/ { this }
  veryBusy(this@UMin(_) -> next) =         veryBusy(next) \/ { this }
  veryBusy(this@Mul(_,_) -> next) =        veryBusy(next) \/ { this }
  veryBusy(this@Div(_,_) -> next) =        veryBusy(next) \/ { this }
  veryBusy(this@Mod(_,_) -> next) =        veryBusy(next) \/ { this }
  veryBusy(this@Add(_,_) -> next) =        veryBusy(next) \/ { this }
  veryBusy(this@Sub(_,_) -> next) =        veryBusy(next) \/ { this }
  veryBusy(this@Not(_) -> next) =          veryBusy(next) \/ { this }
  veryBusy(this@Or(_,_) -> next) =         veryBusy(next) \/ { this }
  veryBusy(this@Eq(_,_) -> next) =         veryBusy(next) \/ { this }
  veryBusy(this@Gt(_,_) -> next) =         veryBusy(next) \/ { this }
  veryBusy(this@Lt(_,_) -> next) =         veryBusy(next) \/ { this }
  veryBusy(this@Geq(_,_) -> next) =        veryBusy(next) \/ { this }
  veryBusy(this@Leq(_,_) -> next) =        veryBusy(next) \/ { this }
  veryBusy(this@Neq(_,_) -> next) =        veryBusy(next) \/ { this }
  veryBusy(this@Cast(_,_) -> next) =       veryBusy(next) \/ { this }
  veryBusy(this@TerIf(_,_,_) -> next) =    veryBusy(next) \/ { this }
  veryBusy(this@FuncCall(_,_,_) -> next) = veryBusy(next) \/ { this }
  veryBusy(this@ProcCall(_,_,_) -> next) = veryBusy(next) \/ { this }
```

Listing A.9 Very Busy Expressions Analysis for GreenMarl

A.2.6 Constant Propagation Analysis Specification for GreenMarl

The full rules for constant propagation in GreenMarl. This showcases how much the analysis is really an abstract interpreter that is indeed just a lifted concrete interpreter.

```
properties
  constProp: CP

property rules

  constProp(prev -> Assign(n, e)) =
    match constProp(prev) with
      | M1(m, v) => M(m \/ {Var{n} |-> v})
      | _ => CP.top

  constProp(prev -> VarRef(n)) =
    addResult(
      constProp(prev),
      getMap(constProp(prev))[Var{n}])

  constProp(prev -> Abs(_)) =
    match constProp(prev) with
     | M1(m, v) => M1(m, constAbs(v))
     | _ => CP.top

  constProp(prev -> UMin(_)) =
    match constProp(prev) with
     | M1(m, v) => M1(m, constUMin(v))
     | _ => CP.top

  constProp(prev -> Mul(_,_)) =
    match constProp(prev) with
     | M2(m, l, r) => M1(m, constMul(l,r))
     | _ => CP.top

  constProp(prev -> Div(_,_)) =
    match constProp(prev) with
     | M2(m, l, r) => M1(m, constDiv(l,r))
     | _ => CP.top

  constProp(prev -> Mod(_,_)) =
    match constProp(prev) with
     | M2(m, l, r) => M1(m, constMod(l,r))
     | _ => CP.top

  constProp(prev -> Add(_,_)) =
    match constProp(prev) with
     | M2(m, l, r) => M1(m, constAdd(l,r))
     | _ => CP.top

  constProp(prev -> Sub(_,_)) =
    match constProp(prev) with
     | M2(m, l, r) => M1(m, constSub(l,r))
     | _ => CP.top
```

Listing A.10 Constant Propagation Property Rules for GreenMarl (1/2)

```
constProp(prev -> Not(_)) =
  match constProp(prev) with
    | M1(m, v) => M1(m, constNot(v))
    | _ => CP.top

constProp(prev -> Or(_,_)) =
  match constProp(prev) with
    | M2(m, l, r) => M1(m, constOr(l,r))
    | _ => CP.top

constProp(prev -> Eq(_,_)) =
  match constProp(prev) with
    | M2(m, l, r) => M1(m, constEq(l,r))
    | _ => CP.top

constProp(prev -> Gt(_,_)) =
  match constProp(prev) with
    | M2(m, l, r) => M1(m, constGt(l,r))
    | _ => CP.top

constProp(prev -> Lt(_,_)) =
  match constProp(prev) with
    | M2(m, l, r) => M1(m, constLt(l,r))
    | _ => CP.top

constProp(prev -> Geq(_,_)) =
  match constProp(prev) with
    | M2(m, l, r) => M1(m, constGeq(l,r))
    | _ => CP.top

constProp(prev -> Leq(_,_)) =
  match constProp(prev) with
    | M2(m, l, r) => M1(m, constLeq(l,r))
    | _ => CP.top

constProp(prev -> Neq(_,_)) =
  match constProp(prev) with
    | M2(m, l, r) => M1(m, constNeq(l,r))
    | _ => CP.top

constProp(prev -> Cast(_,_)) =
  match constProp(prev) with
      // not modelling casts for now
    | M1(m, v) => M1(m, Const.top)
    | _ => CP.top

constProp(prev -> FuncCall(_,_,_)) =
  match constProp(prev) with
    | M2(m, l, r) => M1(m, Const.top)
    | _ => CP.top

constProp(prev -> ProcCall(_,_,_)) =
  match constProp(prev) with
    | M2(m, l, r) => M1(m, Const.top)
    | _ => CP.top

constProp(prev -> _) = constProp(prev)
```

Listing A.11 Constant Propagation Property Rules for GreenMarl (2/2)

```
types
  CPType =
  | M(Map(name, Const))
  | M1(Map(name, Const), ConstProp)
  | M2(Map(name, Const), ConstProp, ConstProp)

  ConstProp =
  | Top()
  | Int(int)
  | String(string)
  | Float(float)
  | Bool(bool)
  | Bottom()

functions
  getMap(cpt: CPType) = match cpt with
  | M(m) => m
  | M1(m,_) => m
  | M2(m,_,_) => m

  addResult(cpt: CPType, v: Const) =
    match cpt with
    | M1(m, v1) => M2(m, v1, v)
    | _ => M1(getMap(cpt), v)

  constAbs(v: Const) = match v with
    | Int(i) =>
      if i < 0
        then Int(-i)
        else Int(i)
    | Float(i) =>
      if i < 0
        then Float(-i)
        else Float(i)
    | _ => Const.top

  constUMin(v: Const) = match v with
    | Int(i) => Int(-i)
    | Float(i) => Float(-i)
    | _ => Const.top

  constMul(l: Const, r: Const) = match (l,r) with
    | (Int(i), Int(j)) => Int(i*j)
    | (Float(i), Float(j)) => Int(i*j)
    | _ => Const.top

  constDiv(l: Const, r: Const) = match (l,r) with
    | (Int(i), Int(j)) => Int(i/j)
    | (Float(i), Float(j)) => Int(i/j)
    | _ => Const.top

  constMod(l: Const, r: Const) = match (l,r) with
    | (Int(i), Int(j)) => Int(i%j)
    | (Float(i), Float(j)) => Int(i%j)
    | _ => Const.top
```

Listing A.12 Constant Propagation Functions for GreenMarl (1/2)

```
constAdd(l: Const, r: Const) = match (l,r) with
  | (Int(i), Int(j)) => Int(i+j)
  | (Float(i), Float(j)) => Int(i+j)
  | _ => Const.top

constSub(l: Const, r: Const) = match (l,r) with
  | (Int(i), Int(j)) => Int(i-j)
  | (Float(i), Float(j)) => Int(i-j)
  | _ => Const.top

constNot(v: Const) = match v with
  | Bool(b) => Bool(!b)
  | _ => Const.top

constOr(l: Const, r: Const) = match (l,r) with
  | (Bool(i), Bool(j)) => Bool(i||j)
  | _ => Const.top

constEq(l: Const, r: Const) = match (l,r) with
  | (Int(i), Int(j)) => Bool(i==j)
  | (Float(i), Float(j)) => Bool(i==j)
  | (String(i), String(j)) => Bool(i==j)
  | (Bool(i), Bool(j)) => Bool(i==j)
  | _ => Const.top

constGt(l: Const, r: Const) = match (l,r) with
  | (Int(i), Int(j)) => Bool(i>j)
  | (Float(i), Float(j)) => Bool(i>j)
  | _ => Const.top

constLt(l: Const, r: Const) = match (l,r) with
  | (Int(i), Int(j)) => Bool(i<j)
  | (Float(i), Float(j)) => Bool(i<j)
  | _ => Const.top

constGeq(l: Const, r: Const) = match (l,r) with
  | (Int(i), Int(j)) => Bool(i>=j)
  | (Float(i), Float(j)) => Bool(i>=j)
  | _ => Const.top

constLeq(l: Const, r: Const) = match (l,r) with
  | (Int(i), Int(j)) => Bool(i<=j)
  | (Float(i), Float(j)) => Bool(i<=j)
  | _ => Const.top

constNeq(l: Const, r: Const) = match (l,r) with
  | (Int(i), Int(j)) => Bool(i!=j)
  | (Float(i), Float(j)) => Bool(i!=j)
  | _ => Const.top
```

Listing A.13 Constant Propagation Functions for GreenMarl (2/2)

```
lattices
  CP where
    type = CPType

    lub(l, r) = match (l,r) with
      | (M(l), M(r)) => M(Map.lub(l,r))
      | (M1(l, cl), M1(r, cr)) =>
          M1(Map.lub(l,r), Const.lub(cl,cr))
      | (M2(l, cl1, cl2), M2(r, cr1, cr2)) =>
          M2(Map.lub(l,r),
             Const.lub(cl1, cr1),
             Const.lub(cl2, cr2))
      | _ => CP.top

    bottom = M(Map.bottom)

    top = M(Map.top)

  Const where
    type = ConstProp

    lub(l, r) = match (l,r) with
      | (Top(), _) => Top()
      | (_, Top()) => Top()
      | (_, Bottom()) => l
      | (Bottom(), _) => r
      | (Int(i), Int(j)) => if i == j then Int(i) else Top()
      | (String(i), String(j)) => if i == j then String(i) else Top()
      | (Float(i), Float(j)) => if i == j then Float(i) else Top()
      | (Bool(i), Bool(j)) => if i == j then Bool(i) else Top()
      | _ => Top()

    bottom = Bottom()
```

Listing A.14 Lattice Definitions for Constant Propagation in GreenMarl

A.3 Reaching Definitions Analysis for Stratego

Named bound-unbound-vars in the Stratego compiler (E. Visser 2013), this code is executed on Stratego core, but—due to legacy reasons—is defined on a larger subset of Stratego. It uses dynamic rewrite rules to encode name binding, mixed with data-flow analysis.

```
module bound-unbound-vars
imports stratego/strc/lib/stratlib
strategies

  mark-bound-unbound-vars =
    mark-buv

  mark-bound-unbound-vars-old =
    if-verbose4(say(!"marking bound-unbound-vars"))
    ; Specification(
        {| MarkVar
         : at-last(Strategies(map(mark-buv)))
         |}
      )
    ; if-verbose4(say(!"marked bound-unbound-vars"))

  /**
   * Annotate variables with one of the annotations "bound",
   * "unbound", or "(un)bound".
   *
   * Variables are bound in matches, used in builds, and
   * refreshed in scopes. Choice operators may lead to
   * a variable being bound in one path, but not in the
   * other. Such variables are annotated with "(un)bound".
   */

  mark-buv = //debug(!"mark-buv in: "); dr-print-rule-set(|"MarkVar"); (
    mark-match
    <+ mark-build
    <+ mark-scope
    <+ mark-let
    <+ mark-traversal
    <+ mark-sdef
    <+ mark-rdef
    <+ mark-lrule
    <+ mark-srule
    <+ mark-overlay
    <+ mark-call
    <+ mark-prim
    <+ mark-rec
    <+ mark-choice(mark-buv)
    <+ mark-lchoice(mark-buv)
    <+ mark-guardedlchoice(mark-buv)
    <+ all(mark-buv)
  //); debug(!"mark-buv out: "); dr-print-rule-set(|"MarkVar")
```

Listing A.15 Reaching Definitions as implemented in the Stratego compiler, written in Stratego using the dynamic rules feature. (1/5)

```
strategies

  DeclareUnbound =
    where(!"unbound" => anno)
    ; ?x
    ; rules(MarkVar+x : Var(x) -> Var(x){anno})

  IntroduceBound =
    where(!"bound" => anno)
    ; ?x
    ; rules(MarkVar+x : Var(x) -> Var(x){anno})

  DeclareBound =
    where(!"bound" => anno)
    ; ?x
    ; rules(MarkVar.x : Var(x) -> Var(x){anno})

  DeclareMaybeUnbound =
    where(!"(un)bound" => anno)
    ; ?x
    ; rules(MarkVar.x : Var(x) -> Var(x){anno})

  undefine-unbound-MarkVar =
    where(map(
      where(<mark-var> Var(<id>) => Var(_){"unbound"})
      ; DeclareMaybeUnbound
    ))

  mark-var =
    bagof-MarkVar; select-mark

  select-mark =
    ?[] < fail + ?[<id>] <+ \ [Var(x) | _] -> Var(x){"(un)bound"} \

  fork-MarkVar(s1, s2) =
    s1 \MarkVar/ s2

strategies

  mark-scope =
    Scope(?xs, {| MarkVar : where(!xs; map(DeclareUnbound)); mark-buv|})

  mark-match =
    Match(mark-match-vars)

  mark-match-vars =
      Var(id)      < MarkAndBind
    + App(id,id)   < App(mark-buv, mark-build-vars)
    + RootApp(id) < RootApp(mark-buv)
    + all(mark-match-vars)

  MarkAndBind =
    try(mark-var)
    ; Var(DeclareBound)
```

Listing A.16 Reaching Definitions as implemented in the Stratego compiler, written in Stratego using the dynamic rules feature. (2/5)

```
mark-build =
  Build(mark-build-vars)

mark-build-vars =
    Var(id)     < mark-var
  + App(id,id)  < App(mark-buv, mark-build-vars)
  + RootApp(id) < RootApp(mark-buv)
  + all(mark-build-vars)

mark-traversal =
  (?All(_) + ?One(_) + ?Some(_))
  ; fork-MarkVar(id, one(mark-buv))

mark-call =
  Call(id,id)
  //; debug(!"call a: ")
  ///; dr-print-rule-set(|"MarkVar")
  ; fork-MarkVar(id, Call(id, mark-buv))
  //; debug(!"call b: ")
  //; dr-print-rule-set(|"MarkVar")

mark-call =
  CallT(id,id,id)
  //; debug(!"callt a: ")
  //; dr-print-rule-set(|"MarkVar")
  ; fork-MarkVar(id, CallT(id, id, map(mark-build-vars)); CallT(id, mark-buv,
 id))
  //; debug(!"callt b: ")
  //; dr-print-rule-set(|"MarkVar")

mark-call =
  CallDynamic(id,id,id)
  //; debug(!"callt a: ")
  //; dr-print-rule-set(|"MarkVar")
  ; fork-MarkVar(id, CallDynamic(mark-build-vars, id, map(mark-build-vars))
  ; CallDynamic(id, mark-buv, id))
  //; debug(!"callt b: ")
  //; dr-print-rule-set(|"MarkVar")

mark-prim =
  PrimT(id,id,id)
  ; fork-MarkVar(id, PrimT(id, id, map(mark-build-vars)); PrimT(id, mark-buv,
 id))
```

Listing A.17 Reaching Definitions as implemented in the Stratego compiler, written in Stratego using the dynamic rules feature. (3/5)

```
mark-let =
  Let(id, id)
  ; where(?Let(<tvars>,_); undefine-unbound-MarkVar)
  ; Let(map(fork-MarkVar( mark-buv
                         , id))
       ,    fork-MarkVar(id, mark-buv))

mark-sdef :
  SDefT(f, as1, as2, s) -> SDefT(f, as1, as2, s')
  where <map(?VarDec(<id>,_) + ?DefaultVarDec(<id>))> as2 => as2'
      ; {| MarkVar :
           <map(IntroduceBound)> as2'
           ; <mark-buv> s => s'
        |}

mark-rdef :
  RDefT(f, as1, as2, r@Rule(t1, t2, s)) ->
  RDefT(f, as1, as2, Rule(t1', t2', s'))
  where <map(?VarDec(<id>,_) + ?DefaultVarDec(<id>))> as2 => as2'
      ; <diff>(<tvars> r, as2') => xs
      ; {| MarkVar :
           <map(IntroduceBound)> as2'
           ; <map(DeclareUnbound)> xs
           ; <mark-match-vars> t1 => t1'
           ; <mark-buv> s => s'
           ; <mark-build-vars> t2 => t2'
        |}

mark-rdef :
  RDef(f, as1, r@Rule(t1, t2, s)) ->
  RDef(f, as1, Rule(t1', t2', s'))
  where <tvars> r => xs
      ; {| MarkVar :
           <map(DeclareUnbound)> xs
           ; <mark-match-vars> t1 => t1'
           ; <mark-buv> s => s'
           ; <mark-build-vars> t2 => t2'
        |}

mark-lrule :
  LRule(Rule(t1, t2, s)) -> LRule(Rule(t1', t2', s'))
  where {| MarkVar :
           <tvars> t1; map(DeclareUnbound)
           ; <mark-match-vars> t1 => t1'
           ; <mark-buv> s => s'
           ; <mark-build-vars> t2 => t2'
        |}
```

Listing A.18 Reaching Definitions as implemented in the Stratego compiler, written in Stratego using the dynamic rules feature. (4/5)

```
mark-srule :
  SRule(Rule(t1, t2, s)) -> SRule(Rule(t1', t2', s'))
  where {| MarkVar :
           // <tvars> t1; map(DeclareUnbound)
           <mark-match-vars> t1 => t1'
           ; <mark-buv> s => s'
           ; <mark-build-vars> t2 => t2'
        |}

mark-overlay :
  Overlay(f, xs, t) -> Overlay(f, xs, t')
  where {| MarkVar :
           <map(IntroduceBound)> xs
           ; <mark-build-vars> t => t'
        |}

mark-rec =
  ?Rec(_, _)
  ; fork-MarkVar(id, Rec(id, mark-buv))

mark-choice(uv) =
  Choice(id, id)
  ; fork-MarkVar(Choice(uv,id), Choice(id, uv))

mark-lchoice(uv) =
  LChoice(id, id)
  ; fork-MarkVar(LChoice(uv,id), LChoice(id, uv))

mark-guardedlchoice(uv) =
  GuardedLChoice(id, id, id)
  ; fork-MarkVar(
      GuardedLChoice(uv,id,id); GuardedLChoice(id,uv,id),
      GuardedLChoice(id,id,uv)
    )
```

Listing A.19 Reaching Definitions as implemented in the Stratego compiler, written in Stratego using the dynamic rules feature. (5/5)

Appendix B

Additional Gradual Type Rules for Stratego

In this appendix we list the complete type rules for those aspects of the language for which we have only shown excerpts in Section 4.4, or left out entirely.

Table of Contents

B.1 Match Terms

Terms in match position are checked against the type of the term they are matched against (Figure B.1). The resulting type can be a refinement, a subtype of the input type, while the environment can contain bindings for new local variables introduced by the match term. Casts that need to be done on subterms are collected in a strategy expression, which others rules will schedule to be executed after the match.

Match terms have the [wildcard], which ignore part of the term. This term is not allowed in a build position. For string literals we check that there is a supertype of `string` that is matched against, and we return the more accurate string type.

Variables in a match term could be bound already, or not yet. However, when the type of a variable is already known, that does not necessarily mean it is already bound. The type information could have been discovered in the output term of a type annotated rewrite rule, and this match could be part of the side-condition of the rewrite rule. Therefore [mvar1] will compute a coercion cast the variable with that coercion after the match. If the variable was previously unbound, this cast will do the required dynamic type check. If the variable was already bound, it was already of the right type and the cast will succeed by default. The [mvar2] rule discovers a new variable and simply records its type.

Constructors can be matched against dynamically and statically typed terms, which is what the [mconstr1] and [mconstr2] rules correspond to. In a dynamically typed context, we only require a constructor of the right arity to exist, and for all the subterms to type-check. In a statically typed context we need a constructor where the subterms can type-check against the types of the constructor definition, and the sort of the constructor is a subtype of the type of the term matched against. This rule is algorithmic as long as it is forbidden to define multiple constructors of the same arity on sorts where one the subtype of another.

Finally, type ascription will test the ascribed term under the type that was ascribed, and then test that this ascription makes sense in the context with a cast in [mascr].

Match term typing $\boxed{\Gamma; t \vdash_m e \Rightarrow e; s \dashv \Gamma; t}$

$$\frac{}{\Gamma_1; t_1 \;\vdash_m \_ \Rightarrow \_;\texttt{id} \dashv \Gamma_1; t_1}\;[\text{wildcard}] \qquad \frac{\Gamma_1 \;\vdash \texttt{string} <: t_1}{\Gamma_1; t_1 \;\vdash_m sl \Rightarrow sl; \texttt{id} \dashv \Gamma_1; \texttt{string}}\;[\text{mstr}]$$

$$\frac{\Gamma_1 \;\vdash t_1 \rightsquigarrow t_2 \;:\; c}{\Gamma_1, x : t_2; t_1 \;\vdash_m x \Rightarrow x; \texttt{<cast(}c\texttt{)>}\; x \dashv \Gamma_1, x : t_2; t_1}\;[\text{mvar1}]$$

$$\frac{}{\Gamma_1; t_1 \;\vdash_m x \Rightarrow x; \texttt{id} \dashv \Gamma_1, x : t_1; t_1}\;[\text{mvar2}]$$

$$\frac{\begin{array}{c} len(\overline{e_1}) = j \qquad \Gamma_1 = \Gamma_1', f_j : \overline{t_1} \rightarrow t_1 \\ \Gamma_1; \texttt{?} \;\; \overrightarrow{\vdash_m} \;\; \overline{e_1} \Rightarrow \overline{e_2}; s \dashv \Gamma_3; \overline{t_2} \end{array}}{\Gamma_1; \texttt{?} \;\vdash_m f\texttt{(}\overline{e_1}\texttt{)} \Rightarrow f\texttt{(}\overline{e_2}\texttt{)}; s \dashv \Gamma_3; \texttt{?}}\;[\text{mconstr1}]$$

$$\frac{\begin{array}{c} len(\overline{e_1}) = j \qquad \Gamma_1 = \Gamma_1', f_j : \overline{t_1} \rightarrow t_2 \\ \Gamma_1; \overline{t_1} \;\; \overrightarrow{\vdash_m} \;\; \overline{e_1} \Rightarrow \overline{e_2}; s \dashv \Gamma_2; \overline{t_2} \qquad \Gamma_2 \;\vdash t_2 <: t_1 \end{array}}{\Gamma_1; t_1 \;\vdash_m f\texttt{(}\overline{e_1}\texttt{)} \Rightarrow f\texttt{(}\overline{e_2}\texttt{)}; s \dashv \Gamma_2; t_2}\;[\text{mconstr2}]$$

$$\frac{\Gamma_1; t_2 \;\vdash_m e_1 \Rightarrow e_2; s \dashv \Gamma_2; t_3 \qquad \Gamma_2 \;\vdash t_1 \rightsquigarrow t_2 \;:\; c}{\Gamma_1; t_1 \;\vdash_m e_1 \;::\; t_2 \Rightarrow x\texttt{@}e_2; (\texttt{<cast(}c\texttt{)>}\; x\texttt{;}\; s) \dashv \Gamma_2; t_3}\;[\text{mascr}]$$

Figure B.1 Algorithmic typing rules for the core Stratego terms in match position. Alternative rules are tried in order.

B.2 Type Inference and Control Flow

We take into account that local variables may be used in different branches of the control-flow with different types, and support such flow-sensitive types:

```
local-variable-type-inference: a -> b
where if <s> a
  then c := <returns-a-list> a
     ; <map(do-something)> c // c :: List(?)
  else c := <returns-a-pair> a
     ; <Snd; do-something> c // c :: ? * ?
  end
; b := <something-else> c // c :: ?
```

Since `something-else` could handle both lists and pairs, this can work out fine. In the future we may explore an extension of the type system with union types to more accurately models such situations. The flow-sensitive typing is realised by the least upper bound operation in the rule for guarded choice (to which the if-then-else construct desugars).

B.3 Strategies

The [fail] rule in Figure B.2 overrides the current type with $\uparrow$, a type compatible with any other type without need for casts. Our least-upper-bound operator is biased against producing this type. The [id] rule passes it's inputs as expected.

The [scope1] and [scope2] rules makes a variable x fresh for the scope of the body. This means it is unbound in the environment for the strategy in the scope. We ignore the type that it is bound to after the strategy, and reinstate the old binding. The [seq] rule sequences by threading the environment and current type through.

The [guardedchoice] rule may appear the same as [seq] rule, but passes the original environment and current type to the s_3 branch. The environments and current types of the s_2 and s_3 branches are merged afterwards with a least-upper-bound. The environments can only differ in local variable bindings at this point, where all bindings are kept and duplicate ones are merged by least-upper-bound.

Strategy typing $\boxed{\Gamma; t \vdash s \Rightarrow s \dashv \Gamma; t}$

$$\frac{}{\Gamma; t \vdash \texttt{fail} \Rightarrow \texttt{fail} \dashv \Gamma; \uparrow}\ [\text{fail}] \qquad \frac{}{\Gamma; t \vdash \texttt{id} \Rightarrow \texttt{id} \dashv \Gamma; t}\ [\text{id}]$$

$$\frac{\Gamma_1; t_1 \vdash_m e_1 \Rightarrow e_2; s \dashv \Gamma_2; t_2}{\Gamma_1; t_1 \vdash \texttt{?}e_1 \Rightarrow \texttt{?}x\texttt{@}e_2\texttt{;}\ s\texttt{;}\ \texttt{!}x \dashv \Gamma_2; t_2}\ [\text{match}] \qquad \frac{\Gamma_1; t_1 \vdash_b e_1 \Rightarrow e_2 \dashv \Gamma_2; t_2}{\Gamma_1; t_1 \vdash \texttt{!}e_1 \Rightarrow \texttt{!}e_2 \dashv \Gamma_2; t_2}\ [\text{build}]$$

$$\frac{\Gamma_1; t_1 \vdash s_1 \Rightarrow s_2 \dashv \Gamma_2, x : t_0'; t_2}{\Gamma_1, x : t_0; t_1 \vdash \texttt{\{}x\ \texttt{:}\ s_1\texttt{\}} \Rightarrow \texttt{\{}x\ \texttt{:}\ s_2\texttt{\}} \dashv \Gamma_2, x : t_0; t_2}\ [\text{scope1}]$$

$$\frac{\Gamma_1; t_1 \vdash s_1 \Rightarrow s_2 \dashv \Gamma_2, x : t; t_2}{\Gamma_1; t_1 \vdash \texttt{\{}x\ \texttt{:}\ s_1\texttt{\}} \Rightarrow \texttt{\{}x\ \texttt{:}\ s_2\texttt{\}} \dashv \Gamma_2; t_2}\ [\text{scope2}]$$

$$\frac{\Gamma_1; t_1 \vdash s_1 \Rightarrow s_3 \dashv \Gamma_2; t_2 \qquad \Gamma_2; t_2 \vdash s_2 \Rightarrow s_4 \dashv \Gamma_3; t_3}{\Gamma_1; t_1 \vdash s_1\ \texttt{;}\ s_2 \Rightarrow s_3\ \texttt{;}\ s_4 \dashv \Gamma_3; t_3}\ [\text{seq}]$$

$$\frac{\begin{array}{c}\Gamma_1; t_1 \vdash s_1 \Rightarrow s_4 \dashv \Gamma_2; t_2 \qquad \Gamma_2; t_2 \vdash s_2 \Rightarrow s_5 \dashv \Gamma_3; t_3 \\ \Gamma_1; t_1 \vdash s_3 \Rightarrow s_6 \dashv \Gamma_4; t_4\end{array}}{\Gamma_1; t_1 \vdash s_1\ \texttt{<}\ s_2\ \texttt{+}\ s_3 \Rightarrow s_4\ \texttt{<}\ s_5\ \texttt{+}\ s_6 \dashv \Gamma_3 \sqcup \Gamma_4; t_3 \sqcup t_4}\ [\text{guardedchoice}]$$

$$\frac{\begin{array}{c} len(\overline{s_1}) = j \qquad len(\overline{e_1}) = k \qquad \Gamma_1 = \Gamma_1', f_{j,k} : \texttt{(}\overline{st}\,\texttt{|}\,\overline{t}\texttt{)}\ t_1 \rightarrow t_2 \\ \Gamma_1; \overline{st}\ \overrightarrow{\vdash_{sa}}\ \overline{s_1} \Rightarrow \overline{s_2} \dashv \Gamma_2 \qquad \Gamma_2; \overline{t}\ \overrightarrow{\vdash_b}\ \overline{e_1} \Rightarrow \overline{e_2} \dashv \Gamma_3 \qquad \Gamma_3 \vdash t_0 \rightsquigarrow t_1 : c \end{array}}{\Gamma_1; t_0 \vdash f\texttt{(}\overline{s_1}\,\texttt{|}\,\overline{e_1}\texttt{)} \Rightarrow \texttt{cast(}c\texttt{);}f\texttt{(}\overline{s_2}\,\texttt{|}\,\overline{e_2}\texttt{)} \dashv \Gamma_3; t_2}\ [\text{call1}]$$

$$\frac{\begin{array}{c} len(\overline{s_1}) = j \qquad len(\overline{e_1}) = k \qquad \Gamma_1 = \Gamma_1', f_{0,0} : \texttt{?} \\ \Gamma_1', f_{j,k} : \texttt{(}\overline{\texttt{?}}\,\texttt{|}\,\overline{\texttt{?}}\texttt{)}\ \texttt{?} \rightarrow \texttt{?}; \overline{\texttt{?}}\ \overrightarrow{\vdash_{sa}}\ \overline{s_1} \Rightarrow \overline{s_2} \dashv \Gamma_2 \qquad \Gamma_2; \overline{\texttt{?}}\ \overrightarrow{\vdash_b}\ \overline{e_1} \Rightarrow \overline{e_2} \dashv \Gamma_3 \end{array}}{\Gamma_1; t_1 \vdash f\texttt{(}\overline{s_1}\,\texttt{|}\,\overline{e_1}\texttt{)} \Rightarrow f\texttt{(}\overline{s_2}\,\texttt{|}\,\overline{e_2}\texttt{)} \dashv \Gamma_3; \texttt{?}}\ [\text{call2}]$$

Figure B.2 Algorithmic typing rules for the core Stratego strategy expressions. Alternative rules are tried in order.

Curriculum Vitae

Jeff Smits

1991	Born September 20th in Moordrecht, the Netherlands
2003 – 2010	Gymnasium diploma *Coornhert Gymnasium* in Gouda
2010 – 2013	BSc in Computer Science (*cum laude*) *Delft University of Technology* in Delft Minor in Robotics
2013 – 2016	MSc in Computer Science *Delft University of Technology* in Delft Honours Programme
2015	Research Assistant *Oracle Labs* in Redwood Shores, CA, USA April – October
2016 – 2023	PhD in Computer Science *Delft University of Technology* in Delft Programming Languages research group
2017 + 2018	Research Assistant *Oracle Labs* in Zürich, Switzerland June – August
2020 – 2023	Postdoctoral Researcher *Delft University of Technology* in Delft Programming Languages research group
2023 – present	Research Software Engineer *Delft University of Technology* in Delft Programming Languages and Algorithmics research groups

List of Publications

Jeff Smits, Toine Hartman and Jesper Cockx (2022). 'Optimising First-Class Pattern Matching'. In: *Software Language Engineering (SLE), Proceedings*, pages 74–83. DOI: `10.1145/3567512.3567519`

Jeff Smits and Eelco Visser (2020). 'Gradually typing strategies'. In: *Software Language Engineering (SLE), Proceedings*, pages 1–15. DOI: `10.1145/3426425.3426928`

Jeff Smits, Gabriël Konat and Eelco Visser (2020). 'Constructing Hybrid Incremental Compilers for Cross-Module Extensibility with an Internal Build System'. In: *The Art, Science, and Engineering of Programming* 4.3, 16. DOI: `10.22152/programming-journal.org/2020/4/16`

Jeff Smits, Guido Wachsmuth and Eelco Visser (2020). 'FlowSpec: A Declarative Specification Language for Intra-Procedural Flow-Sensitive Data-Flow Analysis'. In: *Journal of Computer Languages* 57, 100924, page 39. DOI: `10.1016/j.cola.2019.100924`

Jeff Smits, Gabriël Konat and Eelco Visser (Oct. 2019). 'From Whole Program Compilation to Incremental Compilation: A Critical Case'. In: *Workshop on Incremental Computing (IC 2019)*

Jeff Smits and Eelco Visser (Oct. 2018). *Towards Incremental Compilation for Stratego*. Poster at SPLASH

Jeff Smits and Eelco Visser (2017). 'FlowSpec: declarative dataflow analysis specification'. In: *Software Language Engineering (SLE), Proceedings*, pages 221–231. DOI: `10.1145/3136014.3136029`

iD 0000-0002-8053-8868

Publications during bachelor and master

Jeff Smits (Feb. 2016). 'The Static Semantics of the Green-Marl Graph Analysis Language'. Master's thesis. Delft University of Technology. URL: http://resolver.tudelft.nl/uuid:4f07cbbb-d017-41e8-aba6-8ff0c19f258d

Marieke van der Tuin, A. Bastiaan Reijm, Tim K. de Jong and Jeff Smits (July 2013). 'WebLab project'. Bachelor's thesis. Delft University of Technology. URL: http://resolver.tudelft.nl/uuid:bb2d7a13-1bef-4545-bca0-f2b084a04240

Marc Dekker, Pieter Hameete, Michiel Hegemans, Sebastiaan Leysen, Joris van den Oever, Jeff Smits and Koen V. Hindriks (2011). 'HactarV2: An Agent Team Strategy Based on Implicit Coordination'. In: *Programming Multi-Agent Systems (ProMAS), Revised Selected Papers*. Volume 7217. Lecture Notes in Computer Science, pages 173–184. DOI: 10.1007/978-3-642-31915-0_10

Titles in the IPA Dissertation Series Since 2020

M.A. Cano Grijalba. *Session-Based Concurrency: Between Operational and Declarative Views*. Faculty of Science and Engineering, RUG. 2020-01

T.C. Nägele. *CoHLA: Rapid Co-simulation Construction*. Faculty of Science, Mathematics and Computer Science, RU. 2020-02

R.A. van Rozen. *Languages of Games and Play: Automating Game Design & Enabling Live Programming*. Faculty of Science, UvA. 2020-03

B. Changizi. *Constraint-Based Analysis of Business Process Models*. Faculty of Mathematics and Natural Sciences, UL. 2020-04

N. Naus. *Assisting End Users in Workflow Systems*. Faculty of Science, UU. 2020-05

J.J.H.M. Wulms. *Stability of Geometric Algorithms*. Faculty of Mathematics and Computer Science, TU/e. 2020-06

T.S. Neele. *Reductions for Parity Games and Model Checking*. Faculty of Mathematics and Computer Science, TU/e. 2020-07

P. van den Bos. *Coverage and Games in Model-Based Testing*. Faculty of Science, RU. 2020-08

M.F.M. Sondag. *Algorithms for Coherent Rectangular Visualizations*. Faculty of Mathematics and Computer Science, TU/e. 2020-09

D. Frumin. *Concurrent Separation Logics for Safety, Refinement, and Security*. Faculty of Science, Mathematics and Computer Science, RU. 2021-01

A. Bentkamp. *Superposition for Higher-Order Logic*. Faculty of Sciences, Department of Computer Science, VU. 2021-02

P. Derakhshanfar. *Carving Information Sources to Drive Search-based Crash Reproduction and Test Case Generation*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2021-03

K. Aslam. *Deriving Behavioral Specifications of Industrial Software Components*. Faculty of Mathematics and Computer Science, TU/e. 2021-04

W. Silva Torres. *Supporting Multi-Domain Model Management*. Faculty of Mathematics and Computer Science, TU/e. 2021-05

A. Fedotov. *Verification Techniques for xMAS*. Faculty of Mathematics and Computer Science, TU/e. 2022-01

M.O. Mahmoud. *GPU Enabled Automated Reasoning*. Faculty of Mathematics and Computer Science, TU/e. 2022-02

M. Safari. *Correct Optimized GPU Programs*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2022-03

M. Verano Merino. *Engineering Language-Parametric End-User Programming Environments for DSLs*. Faculty of Mathematics and Computer Science, TU/e. 2022-04