



UvA-DARE (Digital Academic Repository)

Exploratory language development

Discovering the unknown unknowns of language design

Frölich, D.

Publication date

2026

Document Version

Final published version

[Link to publication](#)

Citation for published version (APA):

Frölich, D. (2026). *Exploratory language development: Discovering the unknown unknowns of language design*. [Thesis, fully internal, Universiteit van Amsterdam].

General rights

It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations

If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: <https://uba.uva.nl/en/contact>, or a letter to: Library of the University of Amsterdam, Secretariat, P.O. Box 19185, 1000 GD Amsterdam, The Netherlands. You will be contacted as soon as possible.



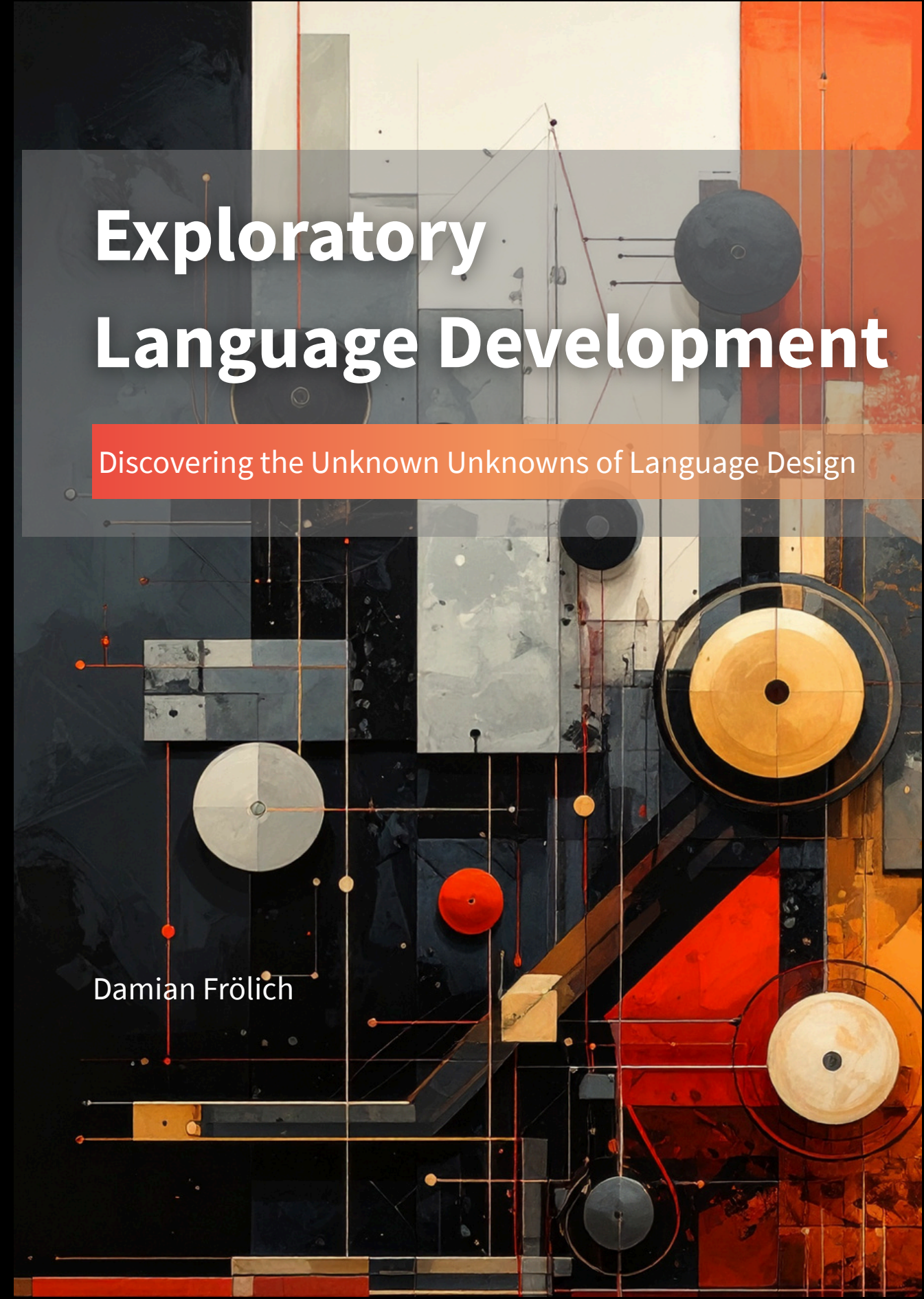
EXPLORATORY LANGUAGE DEVELOPMENT

DAMIAN FRÖLICH

Exploratory Language Development

Discovering the Unknown Unknowns of Language Design

Damian Frölich



EXPLORATORY LANGUAGE DEVELOPMENT

DISCOVERING THE UNKNOWN UNKNOWNNS OF LANGUAGE DESIGN

DAMIAN FRÖLICH

Cover design: Sushmita

This work was carried out in the IPA graduate school.

Copyright ©2025 Damian Frölich

Thesis template: classicthesis

Exploratory Language Development
Discovering the Unknown Unknowns of Language Design

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor

aan de Universiteit van Amsterdam

op gezag van de Rector Magnificus

prof. dr. ir. P.P.C.C. Verbeek

ten overstaan van een door het College voor Promoties ingestelde commissie,

in het openbaar te verdedigen in de Agnietenkapel

op vrijdag 27 februari 2026, te 10.00 uur

door Damian Frölich

geboren te Almere

Promotiecommissie

<i>Promotor:</i>	prof. dr. ir. C.T.A.M. de Laat	Universiteit van Amsterdam
<i>Copromotor:</i>	dr. L.T. van Binsbergen	Universiteit van Amsterdam
<i>Overige leden:</i>	prof. dr. W. Cazzola	University of Milan
	dr. J.A. Baptista Vieira Saraiva	University of Minho
	prof. dr. J.A. Good	Universiteit van Amsterdam
	prof. dr. S. Klous	Universiteit van Amsterdam
	prof. dr. A.W. Goens Jokisch	Universiteit van Amsterdam

Faculteit der Natuurwetenschappen, Wiskunde en Informatica

ACKNOWLEDGMENTS

I would like to start this section by thanking the person who guided me into the direction of a PhD: Ana, you have always been very supportive of my research efforts and ideas. This started while I was a TA and continues to the present day. You *suddenly* asked me if I had interest to do a PhD. After confirming this, you connected me with Thomas, for which I am forever grateful.

The connection with Thomas started somewhere during 2021, and is still going strong. Thomas, thank you for listening to my many ideas and being there throughout my PhD. After the last four years, I can say that I was fortunate with you as a supervisor. You gave me an immense amount of freedom, which I enjoyed and thrived in. But also kept me grounded and focused. You have also given me the opportunity to be more involved with teaching, including giving lectures, which I have enjoyed tremendously. You made me feel involved, but not overwhelmed; independent, but not aimless. Thank you!

To Cees, my promoter, thank you for the expertise and guidance you provided. I enjoyed our meetings, whether it was talking about traveling or about my research. Our discussions helped me view my research from different perspectives. You are meticulous. Having you as my promoter allowed me to build on a solid foundation. Thank you!

To Clemens, thank you for guiding me and having great discussion with me during the last eight years. Thank you for inviting me to Jena, which was a wonderful experience. Hopefully one I can repeat.

To the members in my committee, thank you for accepting the invitation and for reading my thesis.

I want to thank the people in the Agile Language Engineering team for providing a great forum for discussions. Benoit, thank you for inviting us to Rennes and always making the trip an experience.

I want to thank all my colleagues from SNE: Leonardo, Cyril, Na, Yuandou, Saeedeh, Ehsan, Pooya, Misha, Marius, Lu-Chi,

Milen, Daphnee, Romke, Sudaksh, Gabriel, Linus, Cristopher, Tim, Tanjina, Max, Olaf, Eloise, Georgia, Marten, Andrea, Peter, Xin, Tommaso, and many more! Thank you all for the fun moments we shared during the last four years.

To my friends: Floris-Jan, Georgios, Hongyun, Jelle, Kyrian, Marco, Niek, Pieter, Rico, Shashank, Siard, and Wouter. I have enjoyed many wonderful moments with you and look forward to many more to come. To Angelos I can only say one thing: Miep!

To my other friends: Phil, Becky, and Sophie. Throughout the last four years I have been fortunate enough with your company. The game/drinking nights still bring a smile to my face; the dinners we had are a fond memory; showcasing my niche knowledge at the pub-quizzes was great. I am looking forward to the next set of memories we make.

To my family, Mom, Dad, and +Dad: Thank you for always being there for me throughout my life. To my brother: thank you for always being proud of me. To Joy: Thank you for putting up with my brother and joining the family. It is nice to have someone in the family with a similar wavelength as me. To Elina and Lowen: you have been an addition to my life that shifted the way I qualify things as important. I am looking forward to many more sleepovers and watching you two grow up. Ik houd van jullie.

To my Indian family, Mama and Papa, thank you for welcoming me with open arms. I enjoyed my time in India tremendously and I am looking forward to my next visit. To Sushmita and Alan, the trips we made to Bandipur and Turkiye are unforgettable. The next ones will be too.

I end the acknowledgements by thanking my most important person: Rishita. Since I have known you, you have transformed my life. We have made so many memories together in a such a short time, and I look forward to all the memories still to come. Without your support, by proofreading my papers, helping me submit papers at midnight, and just being there for me (to vent), I would have not made it. I therefore dedicate my thesis to you, मेरी जान.

To those I am forgetting at this moment, thank you!

CONTENTS

1	INTRODUCTION	1
1.1	Exploratory Language Development	7
1.2	Research Overview and Contributions	8
1.3	Publications	10
2	EXPLORATORY PROGRAMMING FOR FREE	13
2.1	Introduction	14
2.2	Background	17
2.3	Related Work	19
2.4	A Generic Back-end for Exploratory Programming	20
2.5	A Reusable Architecture for Exploratory Programming	26
2.6	Evaluation & Discussion	34
2.7	Concluding Remarks	49
3	INCREMENTAL LANGUAGE DEVELOPMENT	51
3.1	Introduction	51
3.2	Background	53
3.3	Compositional definitions	58
3.4	Implementation	65
3.5	Examples of incremental compositional language definitions	71
3.6	Discussion	77
3.7	Related Work	81
3.8	Conclusion	83
4	EXPLORATORY LANGUAGE DEVELOPMENT	85
4.1	Introduction	85
4.2	Background	87
4.3	Extended Compositional Definitions	90
4.4	Implementation	94
4.5	A Demonstration of <i>iCoLa</i> ⁺	106
4.6	Discussion	119
4.7	Related Work	128
4.8	Conclusion	132
5	ON THE SOUNDNESS OF AUTO-COMPLETION SERVICES	135
5.1	Introduction	135

5.2	Background	138
5.3	Scope Graphs for Dynamic Languages	146
5.4	Obtaining Scope Graphs via Heaps	152
5.5	Abstract Interpretation with Heap and Frames	158
5.6	Implementation	160
5.7	Experiment Design and Results	161
5.8	Discussion	163
5.9	Related Work	166
5.10	Conclusion	168
6	MULTIVERSE DEBUGGING	169
6.1	Introduction	169
6.2	The Original Multiverse Framework	171
6.3	Grammar and Parser Engineering	174
6.4	Funcons	180
6.5	eFLINT (Reasoning with Norms)	184
6.6	Generalized Multiverse Debugging	188
6.7	Satisfaction of the Requirements	191
6.8	Exploratory Programming and Multiverse Debug- ging	195
6.9	Threats to Validity	196
6.10	Related Work	196
6.11	Conclusion	197
7	CONCLUSION	199
7.1	Contributions	199
7.2	Future work	203
7.3	Final Remarks	206
	BIBLIOGRAPHY	207
	SUMMARY	239
	SAMENVATTING	241

INTRODUCTION

Abstractions are seen in many parts of computer science. For example, networks use the protocol stack to build more complex protocols on simpler ones, where the protocols higher in the stack do not need to understand the implementation details of the protocols below it. Another widely used example is the application of virtual memory as an abstraction over physical memory. Virtual memory enables processes to use more data than can fit in memory by swapping data automatically to disk when required. Virtual memory also provides an isolation layer between processes, therefore increasing security. Such practical abstractions are utilized by programmers without them having to directly interact or even realize the presence of said abstractions. Hiding complexity by introducing abstractions can significantly improve productivity of programmers and the quality of the produced software. One of the most practiced ways in which programmers engage with abstractions are programming languages themselves. The principle abstraction idea behind programming languages is to move away from instructing a CPU directly with CPU opcodes or assembly languages, which are extremely low-level and very error-prone. Nevertheless, the actual abstraction mechanisms observed in programming languages go far beyond abstracting over writing opcodes. This is best illustrated with a brief dive into the history of programming languages as an abstraction mechanism.

PROGRAMMING LANGUAGES AND ABSTRACTION THROUGH A HISTORICAL LENS The usage of programming languages to manage complexity is seen throughout the history of computing, and materializes differently depending on the type of complexity. We utilize this section as a brief exploration into the historical aspects of programming languages and their significance. Note that this section gives a rough overview and does not attempt to cover all interesting aspects of the history of programming

languages. For a complete and detailed treatment of this topic, we refer the reader to the following sources [22, 108, 187, 219].

The origin of programming can be seen as a somewhat physical activity, where big, bulky machines were manipulated to perform calculations. The bulkiness had no effect on the value, as exemplified by the Bombe machine, which helped decode the second variant of the Enigma cipher that was used by the German military to encrypt communication during the second world war [58]. Although the machines remained bulky for a while, the way machines were operated was changing. Originally, programs were written directly in machine code or an octal representation, which was slow and error-prone. This led to the search for better, more efficient methods of programming machines. Early approaches were focused on assembler-like languages that remained close to the machine language but introduced mnemonics that were easier to use than pure machine code. However, with the rise in processing power of machines, the efficiency of constructing programs became more important [219], since less time was spent on waiting for machines to complete calculations. Higher-level languages were investigated to abstract over the complexity of writing programs directly in machine code. An early high-level language that moved away from a machine-like language was Short Code [108]. Short Code encoded mathematical expressions in a more natural way compared to a direct encoding in machine language. Short Code programs were evaluated (interpreted) by a program (interpreter) on a machine. This made the execution of Short Code programs much slower than the same program written directly in machine code. This impact on performance made using Short Code less appealing. Therefore, researchers began looking at programs that could translate programs from a higher-level language to machine code, which was then executed such that the impact on the performance of the program was minimal. At the time, this research direction was often referred to as automatic programming or automatic coding. These days, we call such a program a compiler.¹ Early compilers were developed for a number of languages, such as Whirlwind, FORTRAN, Algol, and COBOL [81, 108, 187]. Next to the advantage of enabling a

¹ The term compiler was coined by Grace Hopper to refer to what we now call a linker [179, 219].

higher-level language to be used to write programs, a compiler could also make programs (software) less dependent on the machine (hardware) that it was originally written for, thus enabling reuse of programs.

COBOL is particularly interesting since it is one of the first languages that aimed to make programming available to a wider audience like businessmen or corporate employees, by focusing less on mathematical notation and using more of an English-like notation. COBOL thus functioned as an abstraction mechanism over the (mathematical) expertise required to program in other higher-level languages. This idea further developed into domain-specific languages (DSLs), where languages are specifically developed for certain domains. This enables the language to integrate domain-specific constructs, enforce domain-specific constraints, and enable domain experts to solve their domain-specific problems. Domain-specific languages have been defined for a plethora of domains, some of which are the legal domain [27, 46], the domain related to writing games [184, 207], and the domain of programming languages itself [15, 67, 105].

A language that was introduced around the same time as COBOL was LISP [128]. LISP differentiated itself from the other languages introduced around that time by taking ideas from the lambda calculus and providing a more functional style to programming, in contrast to the procedural or numerical style seen in other languages at the time. LISP was also the first language to introduce a garbage collector [1, 70], removing the need for programmers to manage memory manually. Along with removing this responsibility of managing memory manually, LISP also reduced memory-related bugs. To this day, such bugs still cause security problems in languages that require manual memory management. Utilizing programming languages to increase the safety and stability of software further developed with, for example Ada [16], and more recently with the Rust language [127].

Although the early higher-level programming languages increase the abstraction level significantly, resulting in improved productivity of programmers, development of new programming languages has not stopped. New application domains of software drive the development of new languages. An example domain is the World Wide Web, which resulted in languages that integrated

well with the application domain, of which PHP is an example. Other languages introduced new programming paradigms, such as Prolog [9, 109] for logic programming, which abstracts over explicit search through a state space; and Erlang [10] for programming concurrent systems, which abstracts over message passing between such systems. Programming languages thus function as abstraction mechanisms over different types of complexity.

THE 101 OF BUILDING A PROGRAMMING LANGUAGE With a somewhat simplified view, two components are needed to build a (new) programming language: a parser and an evaluator. The parser transforms syntax — which the programmer writes — into an internal representation. The evaluator takes this internal representation and assigns behavior to the program. An evaluator can do this by interpreting the program or by compiling it. The parser and evaluator are not trivial to write, but also not impossible. With the right meta-language — the programming language which will be used to implement this (new) programming language — a rudimentary implementation of a language can be obtained with some amount of effort. However, this rudimentary implementation would likely not be sufficient in several aspects. A direct implementation of the conceptual semantics will give substandard performance, hampering the applicability of the language. Only having a parser and an evaluator is not ideal for programmers, who are often supported by auxiliary tools, such as debuggers, profilers, and editor services. Debuggers help programmers with bug discovery in programs by making the program steppable such that intermediate states of a program can be inspected. Profilers help programmers identify performance bottlenecks in their programs both for execution speed and memory usage. Editor services help programmers with the task of programming itself and maintaining the software. Editor services are integrated within text editors or development environments. They provide different functionality, such as auto-completion services, which suggest completion candidates to the programmer; go-to definition services, which simplify navigation through a code base; and refactoring services, which simplify code refactoring. Such auxiliary tooling requires additional engineering efforts, which could become significantly time-consuming, espe-

cially if the aim is to achieve reasonably good performance for both the language implementation and the auxiliary tooling.

TOOLING FOR PROGRAMMING PROGRAMMING LANGUAGES

To support the development of (new) programming languages, tooling that aids various aspects of the construction of programming languages can be used. For example, the Truffle framework takes in an Abstract Syntax Tree (AST)-based interpreter and optimizes the interpreter using several techniques including Just-In-Time (JIT)-compilation. Building a AST-based interpreter is relatively easy compared to the work required to obtain optimizing interpreters, which Truffle gives essentially for free. On the compiler side, the Low Level Virtual Machine (LLVM) framework can be used to reduce the work needed to obtain an optimizing compiler. With LLVM, only a front-end of a compiler is needed, which translates a language into the intermediate representation of LLVM. LLVM then does the optimizations, significantly reducing the work required to obtain an optimizing compiler.

Obtaining tooling for a language can be done using meta-languages or language workbenches. Meta-languages are languages used to process (other) languages. General-purpose languages can function as a meta-language, of which ML (MetaLanguage) [77] and its variants are good examples. DSLs can also be meta-languages. An example of this is Rascal [105], which has built-in constructs to quickly traverse ASTs. Language workbenches provide full-fledged IDE-like environments for the management, creation, and analysis of (domain-specific) languages. Language workbenches are often comprised of multiple meta-languages, of which Spoofox [94] is a good example.

Meta-languages and language workbenches often generate tooling from declarative specifications, significantly reducing the required engineering effort. They also often support the specification of the concrete syntax and semantics of a language, making them a one-stop-shop for the implementation of new programming languages. Some meta-languages and language workbenches focus on the modular construction of programming languages using reusable components. Since programming languages often share constructs, these do not need to be re-implemented, thus reducing the total engineering effort.

A plethora of meta-languages and language workbenches exist. Some of these will be investigated more thoroughly throughout this thesis by relating them to the contribution made in this thesis.

THE COMPLEXITY OF LANGUAGE DESIGN Although meta-languages and language workbenches provide a way to reduce the engineering effort required to build a programming language that also has the necessary auxiliary tools to support programmers, building a new programming language is not just an engineering effort. It also entails a fairly complicated design effort. Many decisions need to be made: does the language embed an existing paradigm? Does it mix paradigms? Is it statically or dynamically typed? Is it strongly or weakly typed? What abstraction mechanisms does it offer (e.g. how can new datastructures be defined)? Does it need to be Turing complete? And so on. These examples are decisions that can be known beforehand, also known as known unknowns. However, there may be design considerations that the designer might not know of yet. For example how different constructs might interact, or how a new feature, which the designer has not thought of yet, can be included in the language. Such unknown unknowns can have a negative effect on the applicability of a language and on the adaptability of a language to new requirements. The difficulty of navigating these choices is exemplified by most modern programming languages. Take for example the shift from Python 2 to Python 3. The newer version had to introduce breaking changes due to design decisions made in the earlier version, such as the original choice to make `print` a keyword, which was changed in favor of a standard function in Python 3. Another example of the difficulty of language design is visible in the JavaScript language where a proposal was made to extend the language with records and tuples. However, due to certain design decisions made for the language, this extension is not possible without making significant changes to the semantics of the language. It required SAT solving to show that there is no design possible that does not require a breaking change to the semantics of the language [180]. Furthermore, design decisions also have an inherent cost-vs-benefit trade-off. This can be a trade-off between complexity and expressiveness of

the language, or between language features and implementation effort, for example. The actual effect of certain decisions on these trade-offs is not always immediately clear.

1.1 EXPLORATORY LANGUAGE DEVELOPMENT

As a way to reduce the combined difficulty of navigating design decisions and experimenting with new design decisions on an existing language, this thesis proposes exploratory language development, which combines language engineering with exploratory programming as a way to navigate and experiment with (design) trade-offs early in the engineering process, while also promoting experimentation with new language features. Central to exploratory language development is the construction of and experimentation with language variants. These variants are used to experiment with design decisions. This constitutes an iterative style of development, where language variants are created, experimented with, and the observations made during the experimentation feedback into the design of new language variants. Exploratory programming is the paradigm we utilize to support this effectively. Exploratory programming is a paradigm where the goal worked towards is open-ended, and by experimenting with code this goal becomes more concrete. This constitutes a volatile style of development where code is easily created and easily discarded. Exploratory language development makes this volatility a central part of the development process. Figure 1.1 gives a visual view of the idea of exploratory language development as presented in this thesis. We distinguish between two types of users: language developers and program developers (programmers). Language developers develop new languages, and program developers write programs in these languages. These two user types are fluid, where one user can function as both types in the same session and switching between user type should be smooth. This setup aligns more with the viewpoint that software engineers are language engineers [222], but does not exclude the other viewpoint. At both levels, the user interacts with an Exploratory Programming Environment (XPE). Languages defined at the meta-level can be experimented with at the object-level. This process is supported by tools that help the

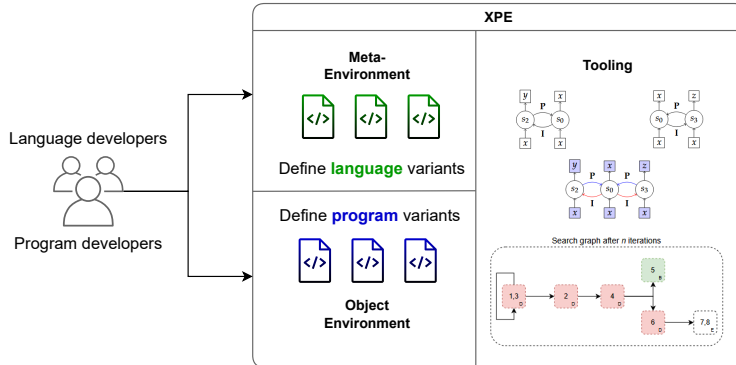


Figure 1.1: An overview of Exploratory Language Development and the different components that support this style. Language developers and Program developers can interact with each other, and a developer can function as both roles. Both activities are supported by an Exploratory Programming Environment (XPE).

construction of substantial programs to better evaluate language variants.

1.2 RESEARCH OVERVIEW AND CONTRIBUTIONS

We give an overview of this thesis according to Figure 1.1. The users interact with XPEs at both the meta-level and the object-level. The goal of the work set out in this thesis is to make it trivial to switch between the meta-level and the object-level, with support for exploratory programming at both levels. Therefore, it must be straightforward to obtain XPEs for both the language used at the meta-level, and the languages defined at the meta-level and used at the object-level. To this end, our goal is to obtain XPEs for free and show that this can be achieved for languages that satisfy certain conditions in Chapter 2.

Reiterating our earlier point, we see variant creation as the primary method for exploring and experimenting with design decisions for programming languages. Although exploratory programming in itself provides the means to achieve this, we aim for a more fine-grained support for language variant creation by

integrating variant creation in the design of a meta-language in Chapter 3 and Chapter 4.

When a variant gets committed, a programmer can experiment with the defined language. To support writing substantial programs during this evaluation, the experimentation should be supported by tools. Again, to easily obtain an environment for the defined language, it must be relatively simple to obtain such tools. If a language engineer has to define tooling for every defined language, it can demotivate the creation of and experimentation with variants. Therefore, our goal is to obtain tooling for defined languages for free, again assuming the language satisfies certain assumptions. In Chapter 5, we look at a reusable approach to obtain sound auto-completion services for dynamically typed languages, and in Chapter 6 we look at obtaining multiverse debuggers for free.

We end this thesis with the conclusions in Chapter 7, where we also shed some light on the future development opportunities in the area of exploratory language development.

CONTRIBUTIONS We make the following high-level contributions in this thesis. In every chapter we give a more detailed account of the specific contributions made in that chapter.

- In Chapter 2 we develop an approach with which a language automatically obtains an exploring interpreter — an interpreter supporting exploratory programming. We combine this with a reusable architecture that decouples interfaces for exploratory programming from runtimes supporting exploratory programming. This decoupling promotes reuse: a user-interface is implemented in terms of an API, which then makes it usable for any runtime implementing this API.
- We introduce a new meta-language with native support for variant creation. Variant creation is achieved by a new take on abstract syntax definitions, and by reusing existing language definitions through support for language composition. This contribution is split into two chapters. We present two implementations, one as an EDSL in Haskell (Chapter 3), and one as an external DSL (Chapter 4).

- In Chapter 5 we present a reusable technique to obtain auto-completion services for dynamically typed languages for free from the operational semantics of the language. Our technique uses a novel combination of abstract interpretation and the scope graph framework to obtain an over-approximation of the name binding seen at run time. We contribute several extensions to the scope graph framework to handle over-approximated scope graphs. With these extensions, we can ensure that completion candidates do not introduce erroneous program flows.
- In Chapter 6 we define debugging user stories and utilize those user stories to collect requirements for multiverse debuggers. Based on these requirements we propose a new framework for multiverse debugging. Our framework generalizes an existing framework by storing more complex histories and supporting different search strategies through the state space.

1.3 PUBLICATIONS

The contributions in this thesis are supported by the following publications.

Chapter 2 **Damian Frölich** and L. Thomas van Binsbergen. “A Generic Back-End for Exploratory Programming.” In: *Trends in Functional Programming - 22nd International Symposium, TFP 2021, Virtual Event, February 17-19, 2021, Revised Selected Papers*. Ed. by Viktória Zsóka and John Hughes. Vol. 12834. Lecture Notes in Computer Science. Springer, 2021, pp. 24–43. DOI: 10.1007/978-3-030-83978-9_2

and

L. Thomas van Binsbergen, **Damian Frölich**, Mauricio Verano Merino, Joey Lai, Pierre Jeanjean, Tijs van der Storm, Benoit Combemale, and Olivier Barais. “A Language-Parametric Approach to Exploratory Programming Environments.” In: *Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2022, Auckland, New Zealand, December 6-7, 2022*. Ed. by Bernd Fischer, Lola Burgueño, and Walter Cazzola. ACM, 2022, pp. 175–188.

DOI: 10.1145/3567512.3567527.

Although not first author on the second paper, the contributions included in this thesis relate to material I contributed to the paper.

- Chapter 3 **Damian Frölich** and L. Thomas van Binsbergen. “iCoLa: A Compositional Meta-language with Support for Incremental Language Development.” In: *Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2022, Auckland, New Zealand, December 6-7, 2022*. Ed. by Bernd Fischer, Lola Burgueño, and Walter Cazzola. ACM, 2022, pp. 202–215. DOI: 10.1145/3567512.3567529
- Chapter 4 **Damian Frölich** and L. Thomas van Binsbergen. “iCoLa+: An extensible meta-language with support for exploratory language development.” In: *Journal of Systems and Software* 211 (2024), p. 111979. ISSN: 0164-1212. DOI: <https://doi.org/10.1016/j.jss.2024.111979>
- Chapter 5 **Damian Frölich** and L. Thomas van Binsbergen. “On the Soundness of Auto-completion Services for Dynamically Typed Languages.” In: *Proceedings of the 23rd ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences. GPCE '24*. Pasadena, CA, USA: Association for Computing Machinery, 2024, pp. 107–120. ISBN: 9798400712111. DOI: 10.1145/3689484.3690734
- Chapter 6 **Damian Frölich**, Tommaso Pacciani, and L. Thomas van Binsbergen. “Exploratory, Omniscient, and Multiverse Diagnostics in Debuggers for Non-Deterministic Languages.” In: *Proceedings of the 18th ACM SIGPLAN International Conference on Software Language Engineering. SLE '25*. Koblenz, Germany: Association for Computing Machinery, 2025, pp. 134–147. ISBN: 9798400718847. DOI: 10.1145/3732771.3742719

During the PhD, the author also contributed to the following two publications.

- Gwendal Jouneaux, **Damian Frölich**, Olivier Barais, Benoit Combemale, Gurvan Le Guernic, Gunter Mussbacher, and L. Thomas van Binsbergen. “Adaptive Structural Operational Semantics.” In: *Proceedings of the 16th ACM SIGPLAN*

International Conference on Software Language Engineering. SLE 2023. Cascais, Portugal: Association for Computing Machinery, 2023, pp. 29–42. ISBN: 9798400703966. DOI: 10.1145/3623476.3623517

- Tommaso Pacciani, **Damian Frölich**, L. Thomas van Binsbergen, and Chrysa Papagianni. “P4DDG: Data-Dependent Grammars for Packet Specification and Parsing in P4.” In: *Proceedings of the 24th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences. GPCE '25. Bergen, Norway: Association for Computing Machinery, 2025, pp. 54–66. ISBN: 9798400719950. DOI: 10.1145/3742876.3742879*

1.3.1 *Data and Software*

The following open-source data sets and software packages are the result of the work presented.

- The work presented in Chapter 2 is available as a reusable artifact obtainable from <https://github.com/leegbestand/ecoop22-artifacts>.
- The data set of Python programs used in the evaluation in Chapter 5 is available here <https://zenodo.org/records/13628718>
- The remaining software packages are available as Git sub-modules linking them to the right commit at <https://gitlab.com/dfrolich/phd>

EXPLORATORY PROGRAMMING FOR FREE

Central to exploratory language development is exploratory programming, since it supports the user both at the meta-level and the object-level. To support easy experimentation with defined languages it must be effortless to obtain support for exploratory programming. In this chapter we introduce an approach with which language implementations automatically obtain support for exploratory when the implementation satisfies certain pre-conditions, and we contribute a reusable architecture for the construction of exploratory programming environment. With the reusable architecture, language-independent interfaces can be constructed and interfaces can be reused between different implementations of the contributed approach.

Associated Publications

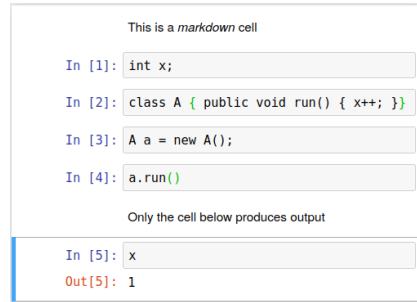
- **Damian Frölich** and L. Thomas van Binsbergen. “A Generic Back-End for Exploratory Programming.” In: *Trends in Functional Programming - 22nd International Symposium, TFP 2021, Virtual Event, February 17-19, 2021, Revised Selected Papers*. Ed. by Viktória Zsóok and John Hughes. Vol. 12834. Lecture Notes in Computer Science. Springer, 2021, pp. 24–43. DOI: 10.1007/978-3-030-83978-9_2
- L. Thomas van Binsbergen, **Damian Frölich**, Mauricio Verano Merino, Joey Lai, Pierre Jeanjean, Tijs van der Storm, Benoit Combemale, and Olivier Barais. “A Language-Parametric Approach to Exploratory Programming Environments.” In: *Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2022, Auckland, New Zealand, December 6-7, 2022*. Ed. by Bernd Fischer, Lola Burgueño, and Walter Cazzola. ACM, 2022, pp. 175–188. DOI: 10.1145/3567512.3567527

```

jshell> int x;
x ==> 0
jshell>
    class A {
        public void run() {
            x++;
        }
    }
| created class A
jshell> A a = new A();
a ==> A@5ce65a89
jshell> a.run()
jshell> x
x ==> 1

```

(a) JShell interaction.



(b) IJava interaction.

Figure 2.1: Example interactions in JShell and IJava.

2.1 INTRODUCTION

Incremental programming is a development style where a program is build incrementally from sequences of smaller programs, and provides an alternative to the traditional compile-edit-run cycle. Incremental programming is often achieved via a Read-Eval-Print Loop (REPL) interpreter. Popular examples of REPLs include JShell for Java, IPython for Python, PsySH for PHP and GHCi for Haskell, which are either part of the language’s distribution (JShell and GHCi) or provide additional features on top of the REPL of the distribution (IPython and PsySH).

Central to REPLs is executing smaller programs and obtaining immediate feedback after the execution of every (intermediate) program. This feedback typically includes the value computed by the program (in case of an expression) and a summary on the (side-)effects of the program, enabling the programmer to update their mental model of the REPLs underlying state. Figure 2.1a demonstrates this via a session in the JShell where a class is defined, after which an object of the class is constructed, and a method is called on the constructed object.

This quicker form of interaction, compared to the compile-edit-run cycle, makes REPLs more suitable for quickly testing library functions, retrieving (type) information on available bindings, experimenting with definitions, debugging, and analyzing data. However, data analysts and other domain-experts, not necessarily skilled in software engineering, prefer to use computational notebooks for these tasks [162, 185]. Computational notebooks are documents consisting of a sequence of three types of cells: code cells, output cells and prose (or documentation) cells. Popular examples are Mathematica [82] and the notebooks built using the Jupyter platform [106]. Code cells are executed one-by-one, with output displayed in output cells, thereby supporting the same kind of incremental program development as REPLs. This is reflected in the design of the Jupyter platform, wherein Python notebooks use the IPython REPL internally [106]. An example of a Jupyter IJava notebook (based on JShell) is given in Figure 2.1b. In the figure, the first cell shows documentation, after which there are several cells containing code and their output, interleaved with some documentation. The numbers before a cell indicate the order in which the cells were executed. Although the examples shows a top to bottom execution order, this is not required by a Jupyter notebook, and cells can be (re)-executed in any order.

REPLs and (Jupyter) notebooks require significant engineering, especially for languages, such as Java and to a lesser extent Haskell, that do not naturally support incremental program development. For example, the code fragments in Figure 2.1 can be recognized as Java code but they do not form a valid Java program individually nor as a sequence. JShell can be seen as implementing an extension of Java rather than Java itself. However, the precise details of this extension – its syntax and semantics – are not clearly specified and are not part of the Java documentation. Moreover, as Figure 2.1 demonstrates, JShell and IJava are not consistent in how they present output. In the example, JShell produces detailed information about the effects of most code fragments whereas IJava only produces output for the last code fragment, revealing a difference between both tools in how they treat computed values and (side-)effects which, one could

argue, are matters of language semantics rather than tool implementation.

A principled approach for the implementation of REPLs, and other interfaces for incremental programming is proposed by van Binsbergen et al. [209]. The approach uses language engineering techniques to explicitly define language extensions, thereby clarifying the difference between the base language and the language implemented by the REPL. The approach makes it possible to develop generic interfaces which under the hood use a definitional interpreter¹ to execute programs. The approach further suggest the use of a so-called exploring interpreter on top of a definitional interpreter for *exploratory programming*. Exploratory programming is an open-ended form of incremental programming in which both the goal and the path towards the goal are discovered as part of the process [21, 172, 205]. The programmer discovers these through interactions with the underlying interpreter by testing definitions, evaluating expressions, analyzing intermediate results and using backtracking to undo work and explore alternative directions.

In the context of programming language engineering, this open-ended form of development can be used at both the meta-level when defining a language, and at the object level when experimenting with an instance of a designed language. To enable this for different language variants during the design phase, supporting exploratory programming for new language variants should be effortless. To that end, in this chapter we contribute a generic implementation of the exploring interpreter algorithm defined by [31] in Haskell, a protocol capturing the essentials of the exploring interpreter algorithm, and a reusable architecture for the construction of exploratory programming environments. The approach is generic in the sense that it can be applied to large class of languages, including all languages that can have their semantics expressed by a transition function, for example in a transition system in the style of Plotkin [164]. By combining the exploring interpreter algorithm with the protocol, we support

¹ A definitional interpreter [176–178] for a language is an interpreter that simultaneously implements and defines the language’s (operational) semantics, often defined in a meta-language or language workbench in the context of domain-specific languages.

the construction of generic programming interfaces that can be reused across languages, and enable experimentation with language-generic functionality for exploratory programming.

This chapter is structured as follows. §2.2 and §2.3 describe background and related work. §2.4 presents an initial implementation of the exploring interpreter algorithm, which is utilized to build a reusable architecture for exploratory programming environments in §2.5. In §2.6 we apply our implementation to three languages – eFLINT, Funcons-beta, and Idris. The resulting specialized exploring interpreters are used in the context of a variety of interfaces. Two of these interfaces have been developed using our reusable architecture. Using the specialized exploring interpreters and connected interfaces, we perform a qualitative evaluation. Based on this evaluation, the initial implementation is extended in several ways in §2.6. Finally, we conclude in §2.7.

2.2 BACKGROUND

This section introduces the methodology and related concepts put forward in [31]. In the proposed methodology, the first step towards developing a REPL for a language is to extend that language to a variant which is in the class of *sequential languages* – the class of languages that naturally support incremental program development. The class of sequential languages is defined in [31] as follows:

Definition 2.2.1. A language L is a structure $\langle P, \Gamma, \gamma^0, I \rangle$ with P a set of programs, Γ a set of configurations, $\gamma^0 \in \Gamma$ an initial configuration and I a definitional interpreter assigning to each program $p \in P$ a function $I_p : \Gamma \rightarrow \Gamma$.

Definition 2.2.2. A language $L = \langle P, \Gamma, \gamma^0, I \rangle$ is *sequential* if there is an operator \otimes such that for every $p_1, p_2 \in P$ and $\gamma \in \Gamma$ it holds that $p_1 \otimes p_2 \in P$ and that $I_{p_1 \otimes p_2}(\gamma) = (I_{p_2} \circ I_{p_1})(\gamma)$.

The chosen definition of languages captures all software language that can have their semantics expressed as a deterministic transition function and includes real-world, large-scale, deterministic programming languages – as demonstrated by the body of literature on big-step, small-step and natural semantics [11,

91, 138, 143, 164] – and further includes languages with non-deterministic aspects when these aspects can be captured algebraically [216]. Configurations capture all information necessary to determine the behavior of a program. A definitional interpreter is thus a program-indexed configuration transformer that assigns to each program (the index) an effect through the transformation on a configuration — a function over configurations. A sequential language is a language in which every sequence of programs is a valid program that has the same effect as the composition of the effects of the individual programs in the sequence.

As an example, consider a simple imperative language such as WHILE [11, 24]. In [24], a transition system is defined to capture the semantics of WHILE commands. A configuration in this system contains a sequence of output values and a store to keep track of variable assignments. The system can be used to give a definitional interpreter for WHILE, as required by Definition 2.2.1, for which it is possible to prove that $I_{C_1;C_2}(\gamma) = (I_{C_2} \circ I_{C_1})(\gamma)$, i.e. to prove that WHILE is a sequential language according to Definition 2.2.2 by choosing ; for \otimes .

The central idea of the approach is that an interpreter for a sequential language can be used, without (further) modification, by the back-end of a REPL, as well as by other interfaces for incremental programming. In other words, a REPL is considered to be just one type of interface for programming in the style that is characteristic of REPLs. The precise behavior of a programming interface is clarified by separating the task of building the interface into language engineering – producing a sequential variant of the base language and an interpreter – and the engineering required to link interface actions to the interpreter and to visualize the effects of programs.

The methodology further proposes the use of a so-called *exploring interpreter* to support exploratory programming. An exploring interpreter is a bookkeeping device on top of a definitional interpreter keeping track of the execution graph — containing the executed programs and visited configurations — while supporting three actions on this graph: **display**, **execute**, **revert**. The **display** action gives a structured representation of (part of) the execution history, to support manipulation of the history or to aid users in building a mental model of the current history. The

execute action of an exploring interpreter for a language executes a program by applying the definitional interpreter for the language while keeping track of the encountered configurations and executed programs in an execution graph, reflecting the entire history of the current interactive session. The execution graph has configurations as nodes and edges between nodes are labeled with programs such that an edge between s and t labeled p indicates that executing p in the context of s yields t , i.e. $I_p(s) = t$. The **revert** action makes it possible to choose any (previously visited) configuration as providing the execution context for the next program, thereby enabling exploratory programming. If the language to which the generic algorithm is applied is a sequential language, then the execution graph of the resulting exploring interpreter is closed under transitivity. This property guarantees the soundness of a variety of operations on the graph.

2.3 RELATED WORK

Definitional interpreters of the kind captured by Definition 2.2.1 can be produced in a language workbench [68] such as the \mathbb{K} framework [117] or Spoofox [95], a meta-language such as Rascal [105], a suitable general-purpose language such as Haskell [86, 126, 160], or can be generated from a formal definition of the operational semantics of the language [13, 28, 196, 212]. These tools and techniques have in common that the semantics of the object language are formulated in an existing (formal) language with well-understood, executable semantics. The first use of definitional interpreters is by Reynolds, employing them as a vehicle for analyzing languages [176, 177]. His analysis took advantage of the formal similarity between denotational and interpretative semantics [178]. Various approaches to formal semantics can be explained in terms of Initial Algebra Semantics [75] in which algebraic signatures denote the constructs of a language and semantics are expressed as algebras over signatures. Modular approaches have been developed that make it possible to extend languages with little or no overhead [199], such as monad transformers [122, 139], algebraic effects and handlers [19, 167, 221], entity propagation in Modular Structural Operational Semantics [13, 143], and copy-rules and forwarding in Attribute

Grammars [198, 210]. These approaches greatly enhance the practice of defining and maintaining definitional interpreters. In modern general-purpose languages, we see advanced use of monads in Haskell [126, 161]; Object Algebras [153] in Java, C# and Scala; intrinsically-typed definitional interpreters in Agda [183]; and embeddings of algebraic effects and handlers in several languages [34, 103, 154]. New languages are also defined where algebraic effects and handlers are native citizens [45, 118].

The usage of an execution graph that contains all configurations produced through program execution is related to back-in-time debugging [32, 120, 124, 168], in which programmers can go ‘back in execution history’. The execution graph, however, captures all components required to reconstruct the full interactive session as it also records the executed programs.

Jupyter is an open-source project for bringing web-based computational notebooks to a wide audience [106]. The Jupyter platform provides a protocol for connecting notebooks to the language kernels, such as IPython and IJava, that take care of program execution. Jupyter is popular and the community supports a large number and wide variety of languages. Within the Jupyter platform, the exploring interpreter algorithm can serve as a layer on top of language kernels to improve support for exploratory programming within Jupyter notebooks.

2.4 A GENERIC BACK-END FOR EXPLORATORY PROGRAMMING

This section presents and discusses a reusable implementation of the exploring interpreter algorithm of [31] in Haskell using a variant of the aforementioned `WHILE` language as an example, of which the encoding is given in listing 2.1.

The definitions of `Command`, `Config`, `initialConfig` and `whileInterpreter` form a language according to Definition 2.2.1. The definitional interpreter (not shown) uses configurations with lists of strings as output and stores to record assignments.

An exploring interpreter is implemented as a parameterized data type, where the type parameters denote the programs and configurations of a given language. The `defInterp` field holds the interpreter responsible for executing programs. The `config`

Listing 2.1: Encoding of the WHILE language in Haskell. The implementation of the interpreter is omitted for brevity, but is available in the supplementary material.

```

data Command = Seq Command Command
             | Assign String Expr
             | Print Expr
             | While Expr Command
             | Skip
data Expr = Leq Expr Expr | Plus Expr Expr
          | LitExpr Literal | Id String
data Literal = LitBool Bool | LitInt Integer

whileInterpreter :: Command → Config → Config
data Config     = Config { cfgStore :: Store, cfgOutput :: Output }
type Store     = Map String Literal
type Output    = [String]
initialConfig   = Config {cfgStore = empty, cfgOutput = []}

```

field stores the current configuration, i.e. the configuration to be used for the execution context of the next program. The `execEnv` field holds the current execution graph and is implemented as an edge-labeled graph using the `fgl` library.² Edges are labeled by programs. The nodes of the execution graph are references (of type `Ref`) to configurations rather than actual configurations. References are implemented as integers and every new configuration gets a unique reference from an increasing counter (using `currRef` and `genRef`). The field `cmap` records the configuration to which each existing reference refers. The field `sharing` determines whether to detect that a configuration has been reached that has already been encountered in which case no fresh reference is generated. With sharing, a configuration is referred to by at most one reference and a node in the execution graph may have multiple incoming edges. Without sharing, multiple references may refer to the same configuration and each node of the execution graph has at most one incoming edge, i.e. the execution graph forms a tree. The `backTracking` field indicates whether a revert action is destructive and deletes nodes and edges.

² <https://hackage.haskell.org/package/fgl>

A smart constructor is defined such that, given a definitional interpreter and an initial configuration, it produces an Explorer.

```
mkExplorer :: Bool → Bool → (p → c → c) → c → Explorer p c
mkExplorer share backtrack interpreter conf = Explorer
  { sharing    = share
  , backTracking = backtrack
  , defInterp  = interpreter
  , config     = conf
  , genRef     = 1
  , currRef    = 1
  , cmap       = IntMap.fromList [(1, conf)]
  , execEnv    = mkGraph [(1, 1)] [] }
```

The smart constructor has additional parameters to determine whether the constructed Explorer should apply sharing and (destructive) backtracking. This gives us a total of four different ways to instantiate the exploring interpreters, resulting in different behaviors of the execution history, as shown in table 2.1. The behavior of the first three configurations are understandable based on the resulting observable type of the execution history. The last instantiation manages the execution history as both a *stack* and a *graph*. Since **revert** is destructive — due to backtracking being enabled — going back to a previous configuration results in deletion of part of the execution graph. As a result, it is not possible to introduce different versions using **revert**, similar to the *stack* configuration. However, because sharing is enabled, variants can be introduced when a configuration is revisited via sharing, hence the *graph* behavior. Since the *stack/graph* configuration is essentially a combination of behaviors seen in other configurations, we decided to focus on the first three types of instantiations of the exploring interpreter, and introduce additional smart constructors for those three configuration types. We discuss the effect of the different ways to model the execution history in more detail in § 2.6.

```
mkExplorerStack = mkExplorer False True
mkExplorerTree  = mkExplorer False False
mkExplorerGraph = mkExplorer True False
```

An Explorer for the WHILE language can then be obtained as follows.

```
type WhileExplorer = Explorer Command Config
whileTree = mkExplorerTree whileInterpreter initialConfig
```

Table 2.1: The different types of behaviors resulting from different instantiations of the exploring interpreter.

Sharing	Backtracking	Type of execution history
\times	\times	<i>stack</i>
\times	\checkmark	<i>tree</i>
\checkmark	\times	<i>graph</i>
\checkmark	\checkmark	<i>stack/graph</i>

Listing 2.2: The Haskell implementation of the **execute** action.

```

execute :: (Eq c, Eq p) => p -> Explorer p c -> Explorer p c
execute p e = updateConf e (p, defInterp e p (config e))

updateConf :: (Eq c, Eq p) => Explorer p c -> (p, c) -> Explorer p
  c
updateConf e (p, newconf) =
  | sharing e =
    case findRef e newconf of
      Just (r, _) ->
        if hasLEdge (execEnv e) (currRef e, r, p)
          then e { config = newconf, currRef = r }
          else e { config = newconf, currRef = r
                  , execEnv = insEdge (currRef e, r, p) (execEnv
                    e) }
      Nothing -> addNewPath e p newconf
  | otherwise = addNewPath e p newconf

```

The exploring interpreter algorithm of [31] describes three actions that can be performed on exploring interpreters: **execute**, **revert** and **display** for executing programs, reverting to previous configurations and displaying the execution graph.

The **execute** action, shown implemented in Haskell in listing 2.2, applies the underlying interpreter on a given program to transition from the current configuration to a (possibly new) configuration. The resulting configuration becomes the current configuration and the Explorer components are updated. If sharing is disabled, a configuration is always seen as unique and a new reference is created, the configuration is added to the execution graph, an edge from the original configuration to the

Listing 2.3: The Haskell implementation of **revert** action

```

revert :: Explorer p c → Ref → Maybe (Explorer p c)
revert e r =
  case IntMap.lookup r (cmap e) of
    Just c | backTracking e → Just e { execEnv = execEnv',
      config = c, cmap = cmap', currRef = r }
    | otherwise → Just e { currRef = r, config = c }
    Nothing → Nothing
where
  nodesToDel = reachable r (execEnv e) \\ [r]
  edgesToDel = filter toDel (edges (execEnv e))
    where toDel (s,t) = s `elem` nodesToDel || t `
      elem` nodesToDel
  execEnv' = (delEdges edgesToDel . delNodes nodesToDel) (
    execEnv e)
  cmap' = deleteMap nodesToDel (cmap e)

```

new configuration is created, and the association between the new reference and configuration is stored. However, if sharing is enabled and the resulting configuration has already been encountered, then the previously assigned reference is used as the target of the new edge.

The **revert** action, shown in listing 2.3, takes a reference and changes the current configuration to the configuration matching the reference. If a reference is given without a corresponding configuration, **Nothing** is returned. If there is a corresponding configuration, then the current reference is changed to the given reference and the current configuration is updated accordingly. Further behavior of **revert** is determined by the *backTracking* field, indicating whether the action is destructive. If it is destructive, then all nodes and edges reachable from the given reference are removed from the execution graph.

Operation **display** produces a structured representation of the execution graph, with the current configuration highlighted. The goal of the display function is to allow interfaces to display and export (parts of) the graph, e.g. to provide an overview, selecting nodes to revert to and saving sessions for later reproduction. To accommodate a wide variety of interfaces, we export several functions for accessing (parts of) the execution graph. For exam-

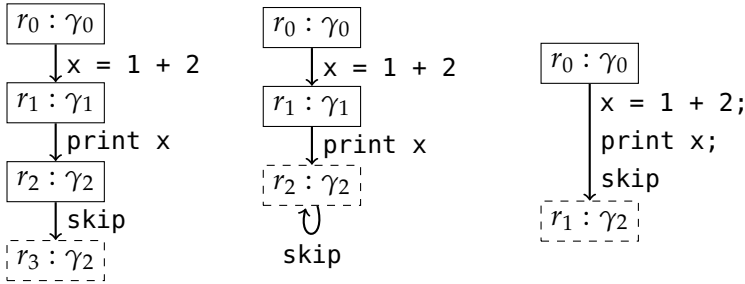


Figure 2.2: Execution graphs after executing the WHILE commands $x = 1 + 2$, $\text{print } x$, and skip without and with sharing, and as a single command respectively. The current node is dashed. The notation $r : \gamma$ denotes a node labeled with reference r referring to configuration γ .

ple, to access the entire execution graph, we export the following function:

```
executionGraph :: Explorer p c → (Ref, [Ref], [((Ref, c), p, (Ref, c))])
```

The result contains the current node, a list of all nodes and a list of all edges in the execution graph. The edges contain both the reference and the referenced configuration of a node.

To obtain only part of the execution graph we export the following functions.

```
getTrace :: Explorer p c → [((Ref, c), p, (Ref, c))]
getTraces :: Explorer p c → [ [((Ref, c), p, (Ref, c))] ]
```

These functions provide one or multiple paths – referred to as traces – from the root node to the current node. As discussed in more detail in the next section, a node might have more than one trace (only) when sharing is enabled.

As an example of using exploring interpreters, consider the following sequence of WHILE commands: $x = 1 + 2$; $\text{print } x$; skip . Figure 2.2 shows the execution graph (with and without sharing) produced when each command in this sequence is executed individually by the exploring interpreter. The first command adds the assignment of literal 3 to identifier x to the store and gives rise to the node with reference r_1 . The second extends the output in the configuration with the literal 3, resulting in the node with reference r_2 . The skip command has no effect on the configuration. Without sharing a new reference is created

nonetheless (reference r_3 on the left of Figure 2.2). With sharing a self-edge labeled with `skip` is created at the node with reference r_2 (middle of Figure 2.2).

FOLDING AND UNFOLDING SEQUENCES The sequence `x = 1 + 2; print x; skip` can also be executed as a single command, resulting in a single edge from r_0 to r_1 (right of Figure 2.2). Because `WHILE` is a sequential language, both interpretations are equivalent in that they yield the same final configuration (γ_2). However, as shown by Figure 2.2, the resulting execution graphs differ significantly, and, depending on the situation, one execution graph might be preferred over the other. Some interfaces might let the programmer determine which execution is chosen. The following function is introduced to offer the flexibility of choice:

```
executeAll :: (Eq c, Eq p) => [p] → Explorer p c → Explorer p c
executeAll = flip (foldl $ flip execute)
```

If a program is a sequence of multiple programs to be executed individually, then then the program can be unfolded to produce a list of programs and executed with `executeAll`. Conversely, if a list of programs is to be executed as a single program, the list can be folded and executed using `execute`.

2.5 A REUSABLE ARCHITECTURE FOR EXPLORATORY PROGRAMMING

The generic back-end discussed in the previous section is Haskell-based, but makes no assumption on the user-interface used to interact with the resulting exploring interpreter. Nor does it provide such an interface. As a result, a language engineer wanting to utilize the resulting exploring interpreter still needs to provide a way for language users to interact with the exploring interpreter. This is contra to our goal: supporting exploratory programming for an object language effortlessly. Therefore, we want an interface to be readily available to an instantiated exploring interpreter without extra effort from the language designer. One way to achieve this, is by reusing interfaces between different languages.

Without a structured approach to building such interfaces around the exploring interpreter, duplicate work is on the horizon. Observed from earlier systems, duplication often occurs at the communication layer between an interface and the system. This problem is also called the $N \times M$ problem, where an interface needs to implement the communication protocol used by M different implementations of the system, in our context an exploring interpreter implementation. When this is repeated for N interfaces, we obtain $N \times M$ implementations that implement communication between an interface and a type of system. This problem is for example observable in the context of editor services, where code editors need to implement a new communication scheme for almost every language. The Language Server Protocol (LSP)³ solves this by giving a standardized communication format between a code editor and a language server. The Debugging Adapter Protocol (DAP)⁴ has a similar function but focused on debuggers and debugger interfaces.

To alleviate this problem in the context of exploratory programming, we propose a reusable architecture built around the generic back-end discussed in the previous section. The architecture is split into a front-end and back-end, which communicate using a protocol. This protocol thus solves the $N \times M$ problem. A visual view of our architecture is given in Figure 2.3. Given a choice of host languages for the front- and back-end, some components of the architecture are reusable, as indicated by the dashed components.

2.5.1 *Exploratory Programming Protocol*

To support communication between an interface and an exploring interpreter, we introduce the Exploratory Programming Protocol (EPP). The EPP is described as a sequence of TypeScript interface definitions, akin to the LSP.

The core of the EPP captures the actions of the exploring interpreter algorithm as RPC-methods and has additional methods to inspect and manipulate the execution history. The full list of methods in the protocol is given in Table 2.2. The *execute*, and

³ <https://microsoft.github.io/language-server-protocol/>

⁴ <https://microsoft.github.io/debug-adapter-protocol>

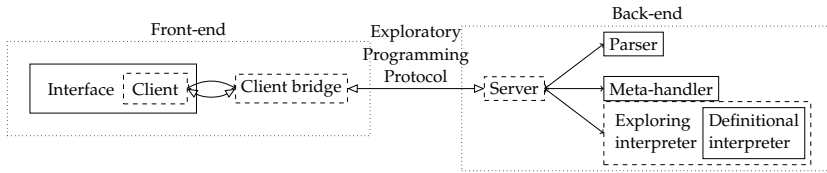


Figure 2.3: The architecture designed for prototyping programming environments with the Exploratory Programming Protocol. Rectangles with dashed lines indicate reusable components given a choice of host languages for the front- and back-end. Solid rectangles are language- or environment-specific components. Arrows with open triangles depict network communication, whereas solid arrows depict function application. Both connection methods are used for the client and client bridge.

revert methods correspond to the actions of the exploring interpreter algorithm. The *getExecutionHistory*, *getTrace*, and *getPath* functions are variants of the **display** action to obtain (parts of) the execution history in a structured format. The *meta* method gives access to meta-commands, providing language-specific services implemented in the back-end that do not involve updates to the execution history. The remaining methods are auxiliary methods to extract information from the execution history, such as the content of a specific configuration or a list containing all nodes without outgoing edges.

2.5.1.1 EPP Specification Using JSON RPC 2.0

The protocol is an instance of JSON RPC 2.0.⁵ The JSON RPC 2.0 protocol defines a request object, a response object, and an error object, which are all encoded as JSON objects. A request object contains an identifier, a method name and the type capturing the parameter(s) of the method (if any). A response object contains an identifier for the request it responds to and either a result or an error. The result can be any encoded JSON object and the error object contains a unique error code, a short descriptive error message, and optional extra error data as an object.

The exploratory programming protocol is an interface between the front-end or GUI of a programming environment and an

⁵ <https://www.jsonrpc.org/specification>

Table 2.2: The methods in the exploratory programming protocol.

Method name	Description
execute	See execute in §2.4 and the protocol specification in §2.5.1.
revert	See revert in §2.4.
getCurrentReference	Gets the reference labeling the current node.
getAllReferences	Returns all references used as a label.
getRoot	Returns the reference labeling the root node.
deref	Gets the configuration assigned to the given reference.
getExecutionHistory	Gets the execution history in the form of the current node a list of edges and list of nodes.
getTrace	Gets the edges representing the path from the root node to the current node.
getPath	Gets the edges representing that path between the nodes labeled by two given references.
getLeaves	Gets the references labeling the nodes without outgoing edges.
meta	Executes a meta-command without affecting the execution tree.

exploring interpreter serving as a back-end. The requests and response pairs of the protocol encode the actions of the exploring interpreter algorithm as JSON objects, of which we detail the **execute** specification in Listing 2.4. The **execute** action is encoded with a request with the method specified as “execute”, and a parameter object containing a string representing the program to be executed.

As a response, the **execute** action can produce an error or a (normal) result, for which the interfaces are defined in Listing 2.5. The result contains the current reference from both before and after the execution, and an optional object containing the result of post-processing the effects of the execution (discussed below). The references and the output are part of the edge added to the

Listing 2.4: Interface definitions for the **execute** action.

```

interface ExecuteRequest extends RequestMessage {
    method: "execute";
    params: ExecuteParams;
}
interface ExecuteParams {
    program: string;
}

```

Listing 2.5: Interface definitions for responses to **execute**.

```

interface ExecuteResponse extends ResponseMessage {
    result?: ExecuteResult;
    error?: ExecuteError;
}
interface ExecuteResult {
    source: uinteger; // reference before execution
    target: uinteger; // reference after execution
    post?: object;
}
interface ExecuteError extends ResponseError {
    code: DefaultErrorCodes | ProgramParseError;
}

```

execution history. The program component completing the edge is part of the request and is omitted from the response.

Following the terminology of §2.2, the *effect* of a program is the set of changes it makes to a configuration when successfully executed. The source and target fields of an `ExecuteResult` object contain all the information necessary to compute the effects of the executed program, using a `DerefRequest` to gain access to the relevant configurations. On top of this, the optional `post` field contains any data that the back-end wishes to send to the front-end in response to an execution request by doing additional post-processing on the execution result. This can be used to compute (a summary of) the effects of a program on behalf of the front-end, as it may be more convenient to compute this information in the back-end. Finally, an **execute** operation might fail, e.g., because the program cannot be parsed (`ProgramParseError`) or the request object is invalid (`DefaultErrorCodes`). In both cases,

the `ResponseError` object contains extra information regarding the error in the forms of a descriptive message and an optional error object giving detailed information.

2.5.2 *Back-end*

The back-end consists of a **server** parameterized by the following (object) language-specific components: a parser, a meta-handler, and a definitional interpreter. The definitional interpreter is used to instantiate a generic exploring interpreter, which maintains the execution history. The server transforms a message from the protocol into operations on the three components. It then takes the result of these operations and transforms them into a message according to the protocol and sends it to the front-end. For example, execute requests are realized via the parser and the definitional interpreter by first parsing the input string with the given parser and then invoking the exploring interpreter, which invokes the definitional interpreter.

Our prototypes are based on a reusable Haskell implementation of the server and the exploring interpreter component introduced in §2.4. The implementation of the `execute` method within the Haskell server is shown in Listing 2.6 (simplified for clarity). The request object is parsed as a JSON object. If the request is not correctly formatted, the `InvalidParameters` error is returned. Otherwise, the parser is applied to the `program` field of the request, returning an error (`Left err`) or a parsed program (`Right prog`). When parsing is successful, the exploring interpreter executes the program, resulting in a possible new configuration. The source and target labels (references) of the (new) edge are part of the result, together with the result of any post-processing.

The **parser** is a language-specific component with the signature: $String \rightarrow c \rightarrow Either\ String\ p$, where c represents the configurations of the language and p the programs. The parser yields either a program or an error, and it has access to the current configuration. Access to the current configuration can be useful for context-sensitive parsing, e.g., Idris allows dynamic extensions of syntax. When the parse is unsuccessful, the parser can provide an error message sent to the front-end as part of the error object.

Listing 2.6: Simplified Haskell fragment of the implementation of `execute` in the back-end server.

```
execute :: Value → ErrorT ErrorMessage (EIP p IO c o) Value
execute v = case (fromJSON v) :: Result ExecuteParams of
  (Error e) → throwError invalidParams
  (Success v_new) → do
    case parse $ program (v_new :: ExecuteParams) of
      Right prog → do
        (ex_new, output) ← execute prog ex
        return $ toJSON $ ExecuteResult
          { source = currRef ex
          , target = currRef ex_new
          , post = postExecute ex ex_new output }
      Left err → throwError ErrorMessage
        { code = programParseErrorCode
        , message = "Supplied program is invalid"
        , error_data = toJSON err }
```

Via the **meta-handler**, a back-end can deliver additional features. A meta-handler has the following signature: `Value -> Explorer p m c o -> m Value`. The handler receives a parameter of the request (a JSON value), the current exploring interpreter, and returns a JSON value. The meta-handler has access to the exploring interpreter to support querying the execution history. This enables a meta-handler to for example search the execution history to find configurations in which a variable or function was defined.

2.5.3 *Front-end*

The front-end is divided into two parts: an interface that extends a reusable client and a bridge that connects the front-end to the back-end.

The **client** provides an API for the interface developer abstracting over communication details. This is achieved by defining the client as an abstract class consisting of concrete methods for performing EPP requests and abstract methods for handling the EPP responses. The concrete methods are implemented once and for all within the client and are generic. These methods assign a unique reference to every request and store the request to be later

Listing 2.7: TypeScript code that shows part of an interface implementation for one of our prototypes.

```
doExecute(input: string) {
  this.execute(new ExecuteParams(input));
}
...
onExecute(req: ExecuteRequest, resp: ExecuteResponse) {
  if (resp.error) {
    this.handleExecuteError(req, resp);
    return;
  }
  showViolations(resp.post);
  ... // additional code making changes in the front-end
}
```

matched with a response. After receiving a response from the client bridge, the client calls the corresponding request handler method. These handler methods are language-specific and must be implemented for every interface. For example, listing 2.7 highlights the handling of **execute** actions in one of our prototypes. A button click triggers the execution of a code cell by calling the handler of the click event *doExecute* which calls the method *execute* of the client by providing the `ExecuteParams` of the request. When the response arrives, the client calls the *onExecute* method with the original request and the response as arguments. The *onExecute* method first determines if the request was successful or not. If the request was successful, the method calls the *showViolations* method to display any violations to the user when any violations are discovered by the back-end's post-processing.

The **client bridge** is an adapter, translating messages from the protocol used between the client and the client bridge into messages of the EPP, and vice versa. This layer of indirection makes it possible to support a wide variety of front-end implementations separate from back-ends. For example, an interface built for use in a web-browser can support exploratory programming via the EPP by communicating using HTTP with a client-bridge, which then translates the HTTP requests into EPP requests, and EPP responses into HTTP responses.

2.6 EVALUATION & DISCUSSION

In this section, we apply our implementation to three languages — eFLINT, Funcons-beta, and Idris — and use the resulting specialized exploring interpreters to perform a qualitative evaluation on the generic implementation and the proposed architecture by connecting the specialized exploring interpreters to two interfaces developed using the reusable architecture. The evaluation investigates the impact of destructive backtracking and sharing on the interactions with the execution history. The result is a discussion on various aspects of exploratory programming, including exploratory programming styles, handling input/output and reproducibility. As part of the evaluation, several extensions to the implementations of the previous sections are discussed.

2.6.1 *Specialized Exploring Interpreters*

We have utilized the generic exploring interpreter to instantiate specialized exploring interpreter for three languages: eFLINT, Funcons-beta, and Idris.

```

#1 > Fact admin                #4 > :session
New type admin                 #1
#2 > +admin(Alice)             |
+admin("Alice")               ` - #2
#3 > +admin(Bob)               |
+admin("Bob")                  +- #3
#4 > :revert 2                 | |
-admin("Alice")                 | ` - #4
-admin("Bob")                   |
#2 > +admin(Bob)               ` - #5
+admin("Bob")                   |
#5 > +admin(Alice)             ` - #4
+admin("Alice")                 #4 >

```

Figure 2.4: A session in the command-line REPL for eFLINT.

The eFLINT language is a DSL for formalizing norms from a variety of sources such as contracts, regulations and business policies [27]. The language currently has three main uses: ex-

ploring a policy specification in order to extend it or improve its internal consistency, statically assessing concrete scenarios for compliance, and dynamically enforcing norms in, and assessing the compliance of, (distributed) software systems. The eFLINT language comes with three existing interfaces to support these tasks, each retrofitted on top of the exploring interpreter for the language, but without using the reusable architecture: a command-line REPL, a web-interface, and a server component supporting remote procedure calls. The language has been extended to a sequential variant by applying the methodology of [31] and the resulting definitional interpreter is used to specialize the generic exploring interpreter developed in this paper. Figure 2.4 shows a simple interaction with the command-line REPL in which a fact-type `admin` is introduced to record admin rights of users. The command-line REPL uses non-destructive reverts and sharing. A configuration contains a knowledge base of facts and after every **execute** and **revert** action the effects on the knowledge base are shown. The `:session` command shows all the traces in the execution graph in the form of a tree.

Figure 2.5 shows a part of the eFLINT web-interface in which a single trace is displayed (obtained via `getTrace`). The web-interface uses destructive backtracking and does not use sharing. The current node therefore has exactly one trace. The web-interface is used by first loading a specification file and then submitting a scenario – a sequence of statements and queries – for execution (using the ‘Send phrase’ button). The effects of statements and queries are shown in green and orange in the displayed trace. Violations are shown in red. A state can be expanded (state 8 in the example) to show the contents of the knowledge base and the last statement in the scenario that produced this state. The buttons below state 8 allow the trace to be updated in various ways by translating button-clicks to combinations of **execute** and **revert** actions.

The server is also used to integrate the specialized exploring interpreter as a reasoning engine in multi-agent and service-oriented systems. Components of such systems can interact with one or more instances of the exploring interpreter to learn dynamically about permissions, obligations and violations. As such,



Figure 2.5: A web-interface for eFLINT showing (part of) a trace. State 8 is expanded.

eFLINT can be used for dynamic policy enforcement and compliance checking.

The PPlanComps project⁶ has identified an open-ended library of so-called fundamental constructs (funcons) that can be used to give a component-based semantics to languages across language paradigms [41, 144]. The funcons have their semantics formally defined in I-MSOS [146] and their I-MSOS specifications are translated to micro-interpreters [24, 28]. These micro-interpreters can be composed arbitrarily to form (definitional) interpreters for different funcon libraries. Funcons-beta is the language defined by the definitional interpreter formed by composing the micro-interpreters of the funcons in the published funcons library.⁷ Figure 2.6 shows the command-line REPL for Funcons-beta built on top of the specialized exploring interpreter for the language. This exploring interpreter is the result of a small language exten-

⁶ <https://plancomps.github.io/>

⁷ <https://plancomps.github.io/CBS-beta/Funcons-beta/Funcons-Index/>

```

#1> bind("input", read)
> "Hello world"
#2> print(bound("input"))
Hello world
#2>

```

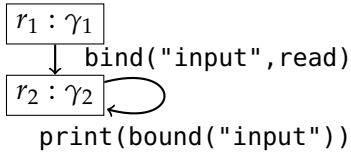


Figure 2.6: A session in the command-line REPL for Funcons-beta.

sion in which Funcons-beta is defined as a sequential language using the *accumulate* funcon as the composition operator \otimes . As a result, bindings produced by executing one funcon term propagate to the next. The first funcon term executed in Figure 2.6 produces a binding for the identifier "input". The second funcon terms prints what was read to *standard-out*.

Idris is a dependently typed functional programming language. With dependent types, types can depend on values, enabling encoding of complex type systems and encoding of many invariants directly into the type system. As a result, these invariants are automatically enforced via type checking. A common example of the usage of dependent types is tracking the length of a vector directly in the type. With this encoding, it becomes impossible to index the vector out of range in correctly typed programs, as shown in Figure 2.7 where calling `sHead` on an empty list results in a type error.

Applying our generic exploring interpreter to these languages required in the order of 50 to 100 lines of Haskell code. Except for Idris, which only requires around 5 lines of code to include error information into the configuration. For both eFLINT and funcons, the main effort was defining the definitional interpreter as an extension of the existing interpreter of the language, which involved carefully choosing the contents of the propagated configuration and the method of handling output. The Idris and Funcons-beta prototypes are especially interesting as they are built on top of existing interpreters developed without anticipating their usage with our approach.

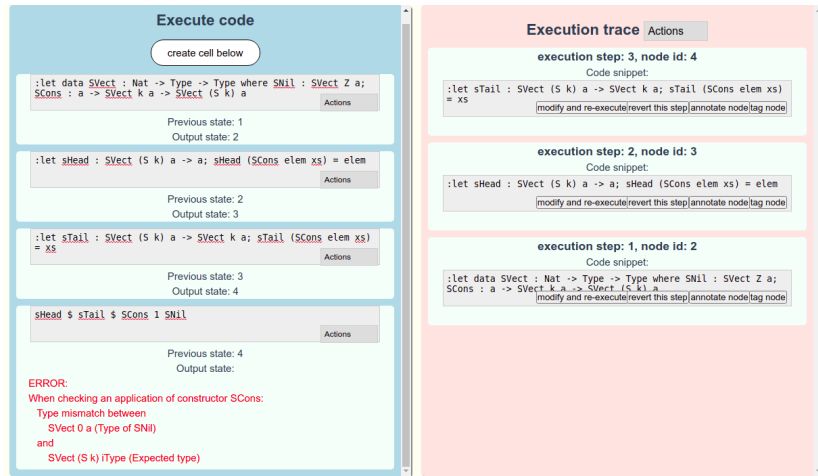


Figure 2.7: Part of an experimental notebook interface with various generic exploratory programming features [111] used with the Idris language. The dependently typed nature of Idris is shown by performing `sHead` on an empty vector, resulting in a type error.

2.6.2 The Shape of Definitional Interpreters

Following the definition of languages (Definition 2.2.1), the specialized exploring interpreters of the previous section consider a definitional interpreter as a pure function expressing the effects of a program by transforming an input configuration.

This approach requires the *simulation* of input and output. For example, output can be considered an ever-growing list of (string) values stored in the configurations, as shown by the definitional interpreter for `WHILE` in §2.4. This choice reduces the potential for sharing since sharing can only take place in between two print statements (discussed further below). Similarly, input can be considered an ever-shrinking list of (string) values with the original input set in the initial configuration.

In Funcons-beta, the *read* funcon reads a value from standard-in as shown in Figure 2.6. In this example, the program `print(bound("input"))` creates a self-edge because output is not part of the configuration and the program has no other effect. The Funcons-beta command-line REPL takes advantage of an implementation of the exploring interpreter algorithm in which

the definitional interpreter can perform effectful computations in a monad, which we present shortly after motivating the design. With this implementation, definitional interpreters have the following type.

```
defInterp :: Monad m => programs -> configs -> m configs
```

The command-line REPL for Funcons-beta instantiates m to the IO monad for interacting with standard-in and standard-out.

The introduction of the monad component has additional advantages. In particular, the monad enables distinguishing between effects and side-effects, with side-effects not being recorded in the execution graph. However, side-effects influence the soundness of the wider approach as the implementation can no longer guarantee that executing a program p in the context of configuration γ yields the same result every time. This has a negative impact on the reproducibility of a session and on the soundness of certain graph operations and optimisations.

The execution trace of Figure 2.5 shows output messages indicating the success of queries and the occurrence of violations. When an eFLINT code fragment is executed (via the ‘Send phrase’ button at the top), the trace can either be updated using DOM manipulation or the page can be refreshed in its entirety. Although not efficient, refreshing is a convenient way to ensure consistency between the front-end and the back-end, as the front-end is redrawn based on the state of the back-end. This then requires the back-end to record output in order to inform the front-end of the output of programs (such as the results of queries) without re-executing programs. However, when output is part of the monad, it is inaccessible for the generic exploring interpreter since its generic in the used monad. To support the reproducibility of output, we have chosen to add an output component to the definitional interpreters:

```
defInterp :: (Monad m, Monoid out) => programs -> configs -> m (
  configs, out )
```

In accordance with Modular Structural Operational Semantics (MSOS) [143, 146], we generalize output to the class of monoidal types, allowing output to concatenate in between executions. Any output produced by the definitional interpreter is stored on the edges of the execution graph, alongside the program producing

that output. The updated definitions of Explorer and execute are as follows:

```
data Explorer p m c o where -- using GADT extension
  Explorer :: (Eq p, Eq c, Monad m, Monoid o) =>
    { defInterp :: p → c → m (c, o), ... } → Explorer p m c o

execute :: (Eq c, Eq p, Monad m, Monoid o) =>
  p → Explorer p m c o → m (Explorer p m c o, o)
execute p e = do (cfg, o) ← defInterp e p (config e)
  return (updateConf e (p, cfg, o), o)
```

As before, the updateConf function is responsible for the extension of the execution graph, now also storing the output on edges. We also extend the ExecuteResult response of the EPP to include the output produced during an execute action.

```
interface ExecuteResult {
  source: uinteger; // reference before execution
  target: uinteger; // reference after execution
  output: object;
  post?: object;
}
```

The new shape handles output in a special way, enabling more features in a generic sense for exploratory interfaces. During our evaluation, we also observed that explicitly encoding erroneous computations provides additional benefits to the implementation of generic interfaces. Currently, one can encode erroneous computations in the definitional interpreter by returning the current configuration, with an optional error component or using the output to communicate errors. However, the resulting effects of such an encoding can be undesirable since a new reference and node in the execution graph is created, possibly confusing users. In addition, interfaces do not have a consistent approach to handle erroneous executions and need to inspect configurations or parse output to visualize erroneous executions. Therefore, we modify the type of definitional interpreters to support failing computations as follows.

```
defInterp :: (Monad m, Monoid out) => programs → configs → m (
  Maybe configs, out)
```

The change wraps the returned configuration inside a Maybe type, which gives the definitional interpreter the option to return Nothing to denote failure, and Just c to denote success resulting

in a new configuration (c). The output component is not wrapped inside the `Maybe` type to support the production of output as part of a failed computation. For example, to collect the output produced by a program before it failed. To handle erroneous computations adequately, we need to again update the `execute` function of the generic exploring interpreter.

```
execute :: (Eq c, Eq p, Monad m, Monoid o) =>
  p → Explorer p m c o → m (Explorer p m c o, o)
execute p e =
  do (mcfg,o) ← defInterp e p (config e)
     case mcfg of
       Just cfg → return (updateConf e (p, cfg, o), o)
       Nothing  -> return (e, o)
```

We also update the `ExecuteError` response object from the EPP.

```
interface ExecuteError extends ResponseError {
  code: DefaultErrorCodes | ProgramParseError | InterpError;
}
```

The `InterpError` code indicates that the `execute` action failed during application of the definitional interpreter. The output produced during this failed execution can be added to either the `error_data` or `message` component of the `ErrorMessage`, or both.

With the updated shape of definitional interpreters, we support a wide variety of languages while supporting specialized handling of output and errors. Nevertheless, the original definition can easily be recovered by instantiating the monad to the identity monad, the output monoid to the empty monoid, and wrapping a pure definitional interpreter with a monadic `return`, which automatically wraps the result in the `Maybe` type. Our implementation of the generic exploring interpreter includes exactly such an implementation, which can be chosen by users instantiating the framework. Alternative variants that instantiate some of the components using an identity component are also imaginable. Hence, the updated definition does not put extra constraints on the definitional interpreters.

2.6.3 On Backtracking and Sharing

DISCUSSIONS ON BACKTRACKING The decision to revert destructively by removing nodes and edges from the execution

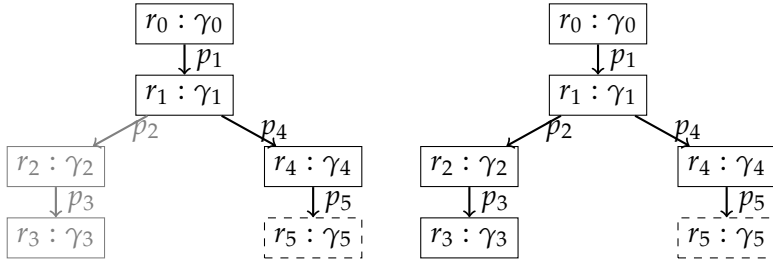


Figure 2.8: Execution graph after execution p_1, p_2, p_3 , reverting to r_1 and executing p_4, p_5 . The gray nodes and edges are removed if the revert action is destructive.

graph has practical and usability-related consequences. Non-destructive reverts enable a more powerful form of exploratory programming. Consider the two execution graphs in Figure 2.8, created with and without destructive backtracking. The figure shows how destructive backtracking ensures that there is always exactly one node in the graph without outgoing edges. In other words, exploration always proceeds along a single path and a revert action always undoes the last n changes along that path (for some n). Conversely, when revert is not destructive, multiple paths are explored simultaneously and strategies like depth-first or breadth-first exploration are possible.

Destructive reverts save space by reducing the size of the execution graph. Applications in which multi-path exploration is not required should, therefore, be able to use destructive reverts. An example of such an application is the execution of a large test-suite in which all tests share a common prefix containing, for example, a number of declarations and initialization statements. In this case, a programmer can execute all tests by executing the prefix once and subsequently executing all tests of the test-suite with backtracking in between tests to undo the changes of the previous test. Executing a test-suite this way can potentially save large amounts of time while the use of space is reduced with destructive reverts. Owing to the implementation presented in this paper, the eFLINT server interface can be used to execute test-suites in the way described.

We conclude that both destructive and non-destructive reverts should be made available to the interface developer on a per application basis. Therefore, we decided to add a new

command to the exploring interpreter: **jump**, which performs a non-destructive revert. In this new model, the **revert** action is always destructive. This gives a programmer the choice to decide when to apply destructive backtracking and when not to. After all, even when multi-path exploration is desired, a programmer might still wish to undo programs.

DISCUSSIONS ON SHARING The decision to apply sharing – i.e. ensuring that every configuration is referred to by at most one node – has significant impact on the practicality and usability of the exploring interpreter. The execution graph is more space-efficient with sharing rather than without, benefiting especially those applications in which output is not stored in configurations (see Figure 2.6 and the discussion on output above). However, detecting opportunities for sharing is costly as it requires comparing (possibly many) configurations for equality. Our implementation determines that the type of configurations used by a language must be an instance of the `Eq` type-class. The `Eq`-instances derived by Haskell compilers use structural equality, a costly operation on large data structures. Moreover, structural equality cannot be used when configurations store functions (such as continuations), in which case a custom equality instance is necessary. This is the case for `Funcons-beta`, in which a function for reading input (using either real or simulated input) is propagated throughout the definitional interpreter. As this function does not change in between calls to **execute**, it is safe to ignore the function when attempting sharing.

Besides space-efficiency, two further advantages of sharing can be observed. Firstly, the exploring interpreter automatically detects the convergence of two exploration paths. In certain applications it will be insightful to the programmer to become aware of convergence. Similarly, sharing will detect cycles. The (abstract) execution graphs of Figure 2.9 give examples of convergence (left) and a cycle (right). The session in Figure 2.4 is a concrete instance of the graph showing convergence in Figure 2.9. Note that by performing effects in a monad, the insights gained from convergence are reduced because convergence only concerns the effects represented by modifications to configurations.

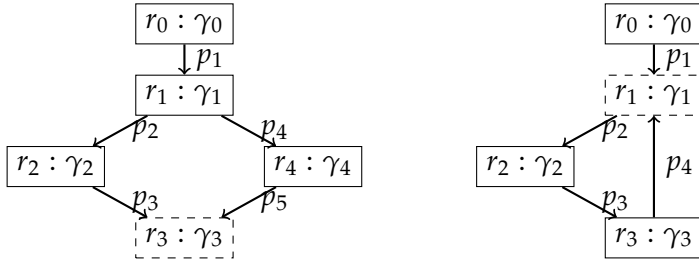


Figure 2.9: Execution graphs showing convergence (left) and a cycle (right).

For the second example of a possible advantage of sharing, consider the situation in the graph on the right-hand side of Figure 2.9 in which r_1 is the current node. If a program p_5 is to be executed next, and if p_5 is equivalent to p_2 , then the exploring interpreter can recognize this and jump to r_2 without executing p_5 (but with adding the edge labeled p_5). If p_5 is a costly program to execute, considerable running time might be saved. This optimization does not depend on sharing; the same situation could arise if the programmer reverted from r_3 to r_1 (without executing p_4 and without destructive backtracking). However, with sharing, opportunities to apply this optimisations are likely to increase in frequency. To further increase the potential of this optimisations it is beneficial to apply normalization techniques to programs. The implementation and analysis of this optimisations is left as future work.

A disadvantage of sharing is that the revert action becomes ambiguous because, with sharing, a node can have more than one incoming edge and more than one trace. Selecting a node in the execution graph is not sufficient to revert to a particular moment in time with a unique history of prior actions. A possible solution is to retain a history of actions in the exploring interpreter. Similarly, it is unclear what the effect of a destructive revert should be in the context of sharing. In the current implementation, all outgoing paths of the new current node are removed from the execution graph.⁸ Sharing also allows cycles that generate infinitely many paths with a repeated infix. These disadvantages demonstrate that sharing significantly complicates the execution

⁸ With the exception of the node itself, in case of a cycle.

graph in a way that makes it harder for the programmer to align their own mental model with the execution graph.

Based on these observations, we have decided to adjust our implementation to always model the execution history as a tree, thus with sharing disabled. To recover some of the benefits of sharing, we provide support to still see when convergence or cycles occur during the exploration session by calculating sharing as a post-processing step on the tree. This functionality is configurable in our final implementation.

2.6.4 An Updated Formal Model

We also update the formal model of [31] to align with the modifications made to the implementation.

Definition 2.6.1. A language L is a structure $\langle P, \Gamma, \gamma_0, O, I \rangle$, with P a set of programs, Γ a set of configurations, $\gamma_0 \in \Gamma$ an initial configuration, O a set of output (values), and I a definitional interpreter that assigns to each program $p \in P$ a function $I_p : \Gamma \rightarrow \Gamma_{\perp} \times O$, where $\Gamma_{\perp} = \Gamma \cup \{\perp\}$ and O is assumed to form a monoid with identity element ε and binary operation \bullet .

Definition 2.6.2. A language $L = \langle P, \Gamma, \gamma_0, O, I \rangle$ is sequential if there is an operator \otimes such that for every $p_1, p_2 \in P$ and $\gamma \in \Gamma$ it holds that $p_1 \otimes p_2 \in P$ and $I_{p_1 \otimes p_2}(\gamma) = (\gamma, \varepsilon) \triangleright I_{p_1} \triangleright I_{p_2}$, where $\triangleright : \Gamma \times O \times (\Gamma \rightarrow \Gamma_{\perp} \times O) \rightarrow \Gamma \times O$, such that $(\gamma, o) \triangleright f = (\gamma \diamond \gamma', o \bullet o')$, where $(\gamma', o') = f(\gamma)$, and $\gamma \diamond \perp = \gamma$ and $\gamma \diamond \gamma'$ is γ' when $\gamma' \neq \perp$.

The new sequential definition extends the original definition with propagation of monoidal values through the sequence of programs and handling of erroneous evaluation, such that whenever the execution of a program does not produce a valid configuration, the sequence continues with the last valid configuration. This definition naturally translates to the behavior seen in many REPLs, where an invalid program does not crash the REPL, instead the REPL continues with the last valid state. Note that the definition does not capture the monad, since the monad is fully abstract from the perspective of the framework, and multiple approaches exist to model computational effects. By leaving

this choice out of the formal model, it is easier to implement the formal model in languages that are less strict regarding the encoding of effects in the type system, such as Java, for example.

Using the new definitions, the definition for the updated exploring interpreter model is as follows.

Definition 2.6.3. An exploring interpreter for a language $\langle P, \Gamma, \gamma_0, O, I \rangle$ is an algorithm maintaining a current reference (initially r_0), a mapping from references to configurations (initially only mapping r_0 to γ_0), and an execution tree (initially only containing the node labeled r_0) upon which the following actions are executed iteratively:

- **execute**(p): let $(\gamma', o) = I_p(\gamma)$, where $p \in P$ is provided as input, $\bar{}$ and γ is the configuration referenced by the current reference. When $\gamma' \neq \perp$, generate a fresh reference r' such that r' maps to γ' , extend the execution tree with the $(r, (p, o), r')$ edge, and transition the current reference to r' .
- **revert**(r): let r' be the current reference. When r is an ancestor of r' in the execution tree, change the current reference to r and remove the (unique) path from r to r' without removing r and any nodes and edges used in other paths from r .
- **jump**(r): When r is a node in the tree, take r as the current reference.
- **display**: provide a structured representation of the exploration tree.

2.6.5 Saving and Loading Sessions

The execution graph of a pure exploring interpreter provides enough information to support the storing and reproduction of sessions generically. One possibility is to export the current configuration, giving the programmer the option to start a new session with the exported configuration as the initial configuration. To also record history, the path from the initial configuration to the current configuration can be exported (i.e. using `getTrace`).

When sharing is enabled, all paths from the root node to the current node can be exported (using `getTraces`).

Exporting paths can be done in two ways affecting in particular the size of the export and the costs of loading a session. The export can contain all components of the path – configurations, references, edges, programs and output – making it possible to load a session without executing programs. However, the soundness of this operation relies on the exploring interpreter being pure; if the programs of the saved session were executed in a monad, then there is no guarantee that the context provided by the monad is the same when the session is loaded (e.g. changes in a database or a file-system). Alternatively, space can be saved by exporting just the sequence of programs labeling the edges on the path. The session can then be loaded by executing this sequence of programs. Assuming the object-language is sequential, this sequence can be folded into a single program as part of the export or as part of loading the session (see the discussion on folding and unfolding in §2.4). Note that in this case, the export is a syntactically valid program that can also be executed with other implementations of the language (e.g. compilers and interpreters).

Finally, the execution graph can be exported in its entirety so that the entire session can be restored.

2.6.6 *Architectural Reuse*

BACK-END The prototypes we developed as instances of the architecture use Haskell implementations of parsers, definitional interpreters, and meta-handlers. Both the server and the exploring interpreter components are language-parametric, and, therefore, only needed to be implemented once. The server and exploring interpreter need to be re-implemented in a different host language to use the protocol and architecture for object languages implemented in that host language. However, this (hypothetical) novel back-end can be combined with existing front-ends, as it relies on the language-agnostic EPP for its communication.

FRONT-END We have two reusable implementations of the client and client-bridge components (hence the two arrows between client and client bridge in Figure 2.3).

1. The first implementation uses the WebSocket protocol as the communication format between the client and client bridge.
2. The second implementation implemented in Python uses native function calls to communicate between the client and client bridge.

A web-based interface was developed independently from us on top of the first implementation [111]. A screenshot of this interface running our Idris case-study is shown in Figure 2.7. We used the second implementation to build a simple native Tk-based interface boosting a notebook-like style. Both front-end implementations were used to develop prototypes on the same Haskell back-end. In general, any implementation of the client (bridge) component can be used in combination with any implementation of the server component.

The **interface** component can be implemented with both features and widgets that are generic and specific to a certain object language. Features can be developed on top of the generic part of the protocol, e.g. executing code in code cells, displaying execution traces, jumping to previous run-time states, etc. Such features are reusable across languages, reducing the workload for language engineers and providing a common experience for programmers switching between languages. On the other hand, a more tailored experience can be offered to programmers with features which are designed specifically for a particular object language, e.g., using post-processing and meta-handlers. With our architecture we can combine generic and language-specific features and replace generic features with specialized variants when available.

One way specialization is achieved is by making the implementation of a feature parametric such that language-specific behavior can be provided as an argument. For example, a variable watcher [132] – showing the assignments to variables in the current run-time state – can be implemented such that a function is given as an argument that extracts variable assignments from

a configuration. A different argument is used for different object languages as each language has its own notion of configuration and approach to keeping track of assignments. Other examples are output cells and visualizations of the execution history when they include information extracted from configurations.

Another approach to specialization is overriding or extending a generic implementation of a feature. The default, generic implementation of the search functionality of our experimental front-end is realized in a text-based fashion by searching the DOM-rendering of the trace. In the Idris prototype, this implementation is replaced with a semantics-based search using the meta-handler for Idris. The semantic search can be used to find only those code cells in a trace in which some variable x is declared or used, whereas with text-based search all occurrences of the letter ' x ' will be found. Another example is for the eFLINT language, in which code cells have been extended with an indicator of any norm violations caused by executing the code cell. When the programmer clicks on the sign, the violations introduced by the program are shown in a modal dialog. In the search example, specialization was realized using the meta-handler for Idris. In the eFLINT example, the specialization was realized by the back-end applying post-processing to extract violations from the configuration produced by executing a program.

2.7 CONCLUDING REMARKS

To support exploratory programming in the context of exploratory language development it must be effortless to support exploratory programming for newly defined object languages. To achieve this, we have built a language-generic implementation of the exploring interpreter algorithm in such a way that any (sequential) language satisfying the precondition of being definable as a transition function over configuration obtains exploratory programming for free. To further support exploratory programming, we proposed a reusable architecture centered around the exploratory programming protocol (EPP). The EPP promotes a consistent communication scheme between a front-end and back-end, reducing duplicate work and enabling independent developments of back-ends and front-ends. We have performed a

qualitative evaluation on the implementation and demonstrated that the generic exploring interpreter can support various styles of exploratory programming, types of interfaces, and types of applications such as command-line REPLs, computational notebooks and servers (e.g. to develop web-applications or multi-agent systems). Furthermore, we have demonstrated that the EPP makes it possible to connect front-ends developed using different hosting environments and implementation languages to independently developed back-ends. As a result, we now have an approach with which we can obtain support for exploratory programming and connect with interfaces specifically designed for exploratory programming for free for any object language satisfying our precondition.

INCREMENTAL LANGUAGE DEVELOPMENT

In the previous chapter we introduced an approach with which an object language can obtain support for exploratory for free by satisfying certain preconditions. In this chapter, we contribute a meta-language that is especially designed to hook into the idea of exploratory programming and the approach laid out in the previous chapter. Concretely, we introduce the meta-language *iCoLa*, focused on incremental programming, and implement this meta-language as an embedded domain-specific language (EDSL) in a mix of Haskell and template Haskell. We demonstrate *iCoLa* through the construction of the Imp, SIMPLE, and MiniJava languages via the composition and restriction of language fragments and demonstrate the variability of our approach through the construction of several languages using a fixed-set of operators.

Associated Publications

- **Damian Frölich** and L. Thomas van Binsbergen. “iCoLa: A Compositional Meta-language with Support for Incremental Language Development.” In: *Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2022, Auckland, New Zealand, December 6-7, 2022*. Ed. by Bernd Fischer, Lola Burgueño, and Walter Cazzola. ACM, 2022, pp. 202–215. DOI: 10.1145/3567512.3567529

3.1 INTRODUCTION

Navigating design decisions for programming languages is complex. Both when defining a new language and when extending an existing language. For example, extending JavaScript with records and tuples cannot be done without introducing a breaking change to the language semantics, which required SAT solving to show [180]. This highlights the complexity of language

design, and the effect early design decisions can have on the later development of a language.

To navigate design decisions, incremental programming can be used. With incremental programming a user repeatedly submits small snippets of code on which they receive immediate feedback, while constructing a larger system via this feedback-loop. As such, incremental programming delivers early feedback on design decisions in the software development process, enabling rapid prototyping and experimentation. In this work we apply incremental programming (also) in the context of language development to support rapid prototyping of languages, enabling a stepwise approach to language design. As such, a language designer can immediately experiment with a defined language variant and feed the observations from this experimentation back into the design. Since languages often have constructs and concepts in common, these can be reused across different language definitions. Reusing existing work further supports rapid prototyping, and thus navigating design decisions.

In this chapter we introduce *iCoLa*, a meta-language focused on the language design process by supporting reusable components and incremental language development to enable rapid prototyping of languages. Concretely, we make the following contributions:

- We present a new approach to defining languages (§3.3) which supports unconstrained composition of languages through incremental programming.
- The approach is implemented as an EDSL with (Template) Haskell as the host language (§3.4). The full power of Haskell is available to define (operations over) languages, achieving the vision of languages as first-class citizens [42] in a general-purpose language.
- We demonstrate reusability and incrementality through case-studies (§3.5) and relate our approach to existing meta-languages by applying Erdweg’s evaluation framework [65] (§3.6).

3.2 BACKGROUND

The approach presented in this chapter combines insights from earlier works to achieve composition of data types. The implementation of the approach is based on certain advanced functional programming techniques described in this section.

INITIAL ALGEBRA SEMANTICS The initial algebra semantics of Goguen et al. [76], concisely described by Mosses in [142], provides the formal foundation and terminology to our work. Initial algebra semantics captures the essential elements of many existing semantic specification formalisms such as denotational semantics and attribute grammars.

A multi-sorted signature (Σ) lays out the operators of a language in terms of a set of sorts. A Σ -algebra assigns carrier sets to these sorts. When taking terms as the carriers by viewing the operators as term constructors, we obtain the abstract syntax of the language. The algebra formed this way is initial in the class of Σ -algebras. Due to its initiality, there is a unique homomorphism from the initial algebra to any algebra in the class of Σ -algebras — also known as a catamorphism [130]. Algebras give meaning to the operators of a signature by assigning a semantic function to each. Following initiality, any abstract syntax can be mapped to the semantics of an algebra in the class of Σ -algebras.

DATA TYPES À LA CARTE As a solution to the expression problem [215], *data types à la carte* [199] provides a method for assembling data types and functions from individual components to form signatures, an initial algebra for every signature, and evaluation algebras, respectively.

With the approach, data types are defined as functors and combined by taking the functor co-product, which is, again, a functor. Using this technique, a simple integer addition language can be defined, which is shown in Listing 3.1, where the `:+` operator implements the functor co-product. The listing also shows the definition of the `Term` data type to support nested expressions and cross-usage of constructs by tying the recursive knot. This is achieved by taking the fix-point of the co-product functor [130].

Listing 3.1: Example showing the usage of the co-product functor to combine two constructs. The Term definition ties the recursive knot.

```

data Val a = Val Int
data Add a = Add a a

type ValAndAdd a = (Val :+: Add) a

data (f :+: g) e = Inl (f e) | Inr (g e)

data Term f = In (f (Term f))

```

Although nested expressions are now supported, we still need to place our expressions on the correct side of our co-product. To automate this, automatic injections into the co-product type via type classes are used.

```

class (Functor sub, Functor sup) => sub <: sup where
    inj :: sub a → sup a

instance f <: f where
    inj = id

instance f <: g where
    inj = Inl

instance (f <: g) => f <: (g :+: h) where
    inj = Inr . inj

```

The `<:` operator defines a typing relation such that if `f <: g`, it means that `f` is subsumed by `g`, i.e. values of type `f` can be constructed as part of values of type `g`.

Since both the data types and the functor co-product are functors, catamorphisms can be used to operate on the composition of data types.

```

foldTerm :: Functor f => (f a -> a) → Term f → a
foldTerm f (In t) = f (fmap (foldTerm f) t)

```

The first argument (highlighted) to the `foldTerm` function is called an algebra, and defines how the resulting value is constructed. To enable the extension of new cases at the function level, the approach implements algebras via type classes. Type

classes are used because they provide ad-hoc polymorphism and are open for extension via instance definitions.

To illustrate the usage of algebras a bit more, we implement an evaluator for our two constructs defined earlier, `Val` and `Add`, which encode integer addition expressions.

```
evalVal :: Val Int → Int
evalVal (Val n) = n

evalAdd :: Add Int → Int
evalAdd (Add e1 e2) = e1 + e2

evalBoth :: (Add :+: Val) Int → Int
evalBoth (Inl f) = evalAdd f
evalBoth (Inr g) = evalVal g
```

The `evalBoth` function can be passed to the `foldTerm` function together with a term containing the `Add` and `Num` constructs. Building such functions and combining them manually is cumbersome. With the help of type classes, we can encode evaluation functions in isolation and derive a combined evaluator automatically. Using type classes, our previous example can be encoded as follows.

```
class Eval f where
  eval :: f Int → Int

instance Eval Val where
  eval (Val n) = n

instance Eval Add where
  eval (Add e1 e2) = e1 + e2

instance Eval f, instance Eval g => instance Eval (f :+: g) where
  eval (Inl f) = eval f
  eval (Inr g) = eval g
```

With this encoding, the combination is automatically derived based on the way the signature is defined, similar to the automatic injections defined earlier. In addition, when adding a new construct to our language, only the instance definition for the new construct needs to be defined. For example, adding a multiplication construct requires the following.

```
data Mult a = Mult a a
instance Eval Mult where
  eval (Mult e1 e2) = e1 * e2
```

The `comp-data` library [14] is a comprehensive Haskell library implementing the data types à la carte approach with some extensions, including support for generalized algebraic data types (GADTs) [89], contexts, and automatically deriving several type class instances using Template Haskell. The library also supports higher-order functors [89] to implement signatures. A consequence of using higher-order functors is that algebras become natural transformations instead of functions, affecting the kind of the algebra and possibly requiring boilerplate code to ensure kind correctness. In Haskell, a kind refers to the type of a type constructor. A type can simply be a constant (an object), denoted with a \star symbol, or a type can be a transformation on object, denoted with the \rightarrow symbol. For example, a functor transforms one type into a new type, and thus has the kind $\star \rightarrow \star$. A higher-order functor has the kind $(\star \rightarrow \star) \rightarrow \star \rightarrow \star$ and thus takes a type transformation (a functor) and a type as arguments, and returns a new type. If we do not pass a functor type as the first argument to the higher-order functor type constructor, the kinds do not match and the application is not kind correct, resulting in an error.

FUNCONS The component-based approach to operational semantics presented in [144] and used as a case study in the previous chapter, is centered around reusable definitions of the *fundamental constructs* of (general-purpose) programming languages – referred to as *funcons* for short. As explained in [29], ‘micro-interpreters’ can be generated from funcon definitions. The micro-interpreters are compositional evaluation functions expressing the behavior of an individual funcon that can be generated and compiled separately. In this chapter, we leverage the generality of the Funcons-beta library [147] to be able to express the semantics of language constructs in a shared base language. Effectively, the generated micro-interpreters for funcons are applied as the constructs of an EDSL to build the funcon terms interpreted by the funcon interpreter.

LANGUAGE COMPOSITION EVALUATION FRAMEWORK Erdweg et al. provide a framework for characterizing and comparing meta-languages, tools, and formalisms that support various

forms of incremental language development [65]. In particular, the authors define the concepts of (modular) language extension, restriction, and unification, which they apply to both the syntax (concrete & abstract), static semantics, operational semantics and IDE services of languages. Extension occurs when a base language is extended by another language that has a dependency on the base language. Restriction is a special form of extension, where a language is restricted, making the new language a subset of the original language. Unification is the process of combining two independent languages with the help of glue code to unify the two languages. The paper also distinguishes between different *forms* of extension: no extension composition, incremental extension, and extension unification. Supporting multiple extensions requires support for extension composition. For incremental extensions, extension can be performed in layers where one extension extends the base and another extension extends the extensions, etc. With extension unification, two extensions are unified and the unification is used as the extension on a base language. In this chapter we adopt their terminology and use their framework as the basis for our evaluation.

TEMPLATE HASKELL Template Haskell is a Haskell extension permitting compile-time meta-programming [197]. With Template Haskell, users can write programs that transform programs. For instance, it is useful when generating boilerplate code or to perform computations at compile-time to improve run-time performance. The extension provides several facilities to inspect and operate on Haskell programs, including a quotation monad that enables reification of Haskell names, giving the programmer access to the internal representation of the compiler. Names are obtained by using prefix quotes, with one quote operating in the expression context and two quotes operate in the type context. The `$` construct is used to evaluate, or splice, a Template Haskell expression. For example, the following splice derives something based on a name obtained from the expression level and a name obtained from the type level: `$(derive 'f "Bool)`. In this example, `f` is a function and its name is obtained by prefixing it with single quote, and `Bool` is a type and its name is obtained by prefixing it with double quotes. These obtained names can

now be reified to get the internal Haskell representation of the underlying constructs, such as `Dec` for declarations and `Exp` for expressions. Functionality such as reifying names and generating fresh names is encapsulated by the `Q` monad, which also functions as an indicator for Template Haskell functions. A Haskell function with a codomain (return type) of `Q Dec` is thus a Template Haskell function that returns a Haskell declaration.

3.3 COMPOSITIONAL DEFINITIONS

In this section we describe our approach to language development conceptually. The insight of incremental language development via composition and the separation between operator (or language construct) definitions on the one hand and language definitions on the other hand, is essential to our approach. A language definition can freely choose from the available operators and constrains the flexibility with which the chosen operators can be used. The definition of an operator consists of an abstract syntax definition and a denotational semantics, choosing functors as a semantic domain. The separation between operator and language definitions is enabled by an alternative take on abstract syntax definitions.

3.3.1 Abstract Syntax

A common approach to defining the abstract syntax of a language is to give algebraic data types (ADTs) of which the operator¹ signatures determine, in a mutually recursive fashion, the set of terms that forms the abstract syntax of the language. For example, the abstract syntax of a lambda calculus can be represented as follows, where Var_O , Abs_O , and App_O are operators (as indicated by the subscript) and $String$ and $Expr$ are sorts.

$$Var_O : String \rightarrow Expr$$

$$Abs_O : String \times Expr \rightarrow Expr$$

$$App_O : Expr \times Expr \rightarrow Expr$$

¹ Such as *constructors* in Haskell and *variants* in the ML family of languages.

In this style, the signature of an operator simultaneously identifies the sort of terms constructed by applications of the operator, the arity of the operator, and the sort of terms required at each operand position in valid applications of the operator.

A key insight of our approach is to delay the decisions related to sorts (but not the arity) until the definition of a language, rather than making these part of operator definitions. This is achieved by (1) using a unique sort at every position in the signature and by (2) introducing separate *sort constraints* to establish the relations between the sorts. Following (1), the sorts are effectively naming operand positions. The right-hand side of a signature is made redundant and can be removed as every operator already has a unique name. With these changes, the operators are defined as follows:

$$\begin{aligned} \text{Var}_{\mathcal{O}} &: \text{VarVar} \\ \text{Abs}_{\mathcal{O}} &: \text{AbsVar} \times \text{AbsBody} \\ \text{App}_{\mathcal{O}} &: \text{AppAbs} \times \text{AppArg} \end{aligned}$$

In contrast to the conventional approach, the signatures do not share any sorts, and the three operators are (as of yet) completely unrelated. To re-establish the relationships, we introduce sort constraints. Sort constraints are based on the interpretation of sorts as sets of operators. For example, the following sort constraint indicates that strings serve as identifiers in both variable references and abstractions:

$$\begin{aligned} \text{String} &\subseteq \text{VarVar} \\ \text{String} &\subseteq \text{AbsVar} \end{aligned}$$

This kind of sort constraint is referred to as a *sub-sort declaration*.

The other kind of sort constraint, referred to as an *operator assignment*, indicates that terms constructed by the $\text{Var}_{\mathcal{O}}$ operator can be used as the body of an abstraction:

$$\text{Var}_{\mathcal{O}} \in \text{AbsBody}$$

To express the same relations between the operators as in the initial example, operator assignments can be written for every pair of an operator and sort taken from the sets $\{\text{Var}_{\mathcal{O}}, \text{Abs}_{\mathcal{O}}, \text{App}_{\mathcal{O}}\}$

and $\{AbsBody, AppAbs, AppArg\}$. Writing down these operator assignments grows increasingly tedious (and error-prone) as more and more operators are added to a language. Therefore, as a convenience, sort constraints can also be used to introduce auxiliary sorts that serve as a level of indirection and enable reuse. The following sort constraints utilize the auxiliary sort $Expr$, stating that all operators assigned to $Expr$ are also assigned to $AbsBody$, $AppAbs$ and $AppArg$:

$$Expr \subseteq AbsBody$$

$$Expr \subseteq AppAbs$$

$$Expr \subseteq AppArg$$

The relations of the original example are then expressed by assigning the operators to $Expr$:

$$Var_{\mathcal{O}} \in Expr$$

$$App_{\mathcal{O}} \in Expr$$

$$Abs_{\mathcal{O}} \in Expr$$

A language designer can introduce new operators with full flexibility and without modifying existing operator definitions because our approach separates operators from constraints detailing where operators can be used. For example, extending the lambda calculus with integer addition can be achieved by defining an Add operator and assigning this operator to the sorts where we want to use the Add operator.

$$Add_{\mathcal{O}} : AddLeft \times AddRight$$

$$Add_{\mathcal{O}} \in Expr$$

This definition adds $Add_{\mathcal{O}}$ to $Expr$, such that the Add operator can be used at the operand positions over which we distributed $Expr$ earlier. Interestingly, no operators have been assigned to the operands of the Add operator yet. Consider the following sort constraints:

$$Integer \subseteq AddLeft$$

$$Integer \subseteq AddRight$$

$$Integer \subseteq Expr$$

$$Add_{\mathcal{O}} \in AddRight$$

These constraints express that integer literals can appear as operands of *Add* in both positions. However, since the *Add* operator is only added to *AddRight*, the constraints allow only nested occurrences of *Add* on the right side, encoding right-associativity. This example demonstrates the flexibility of sort constraints: integer expressions can be used in lambda-expressions — owing to the constraints $Add_{\mathcal{O}} \in Expr$ and $Integer \subseteq Expr$ — whereas lambda-expressions cannot be used in integer expressions. Such rules of composition can be changed simply by selecting a different set of sort constraints without affecting the definitions of the operators themselves. As discussed in §3.3.4, selecting sort constraints is done as part of a language definition.

3.3.2 Compositional Semantics

To retain the disjoint property of the operators, their semantics must be defined independently as well. This is achieved by defining semantic functions that together form an algebra. Semantic functions translate an operator into a specific semantic domain. For example, our previous operators defining the lambda calculus can have the following semantic functions, with funcons being our semantic domain.²

$$\begin{aligned} Var_{\mathcal{F}}(lit) &= \mathbf{bound\ string\ } lit \\ Abs_{\mathcal{F}}(x, b) &= \mathbf{function\ closure\ scope}(\mathbf{bind(string\ } x, \mathbf{given}), b) \\ App_{\mathcal{F}}(abs, arg) &= \mathbf{apply}(abs, arg) \end{aligned}$$

Through the catamorphism, the operands of an operator are already translated by their respective translation function when an operator is translated. Hence, an operator only needs to translate itself into the semantic domain while having access to the already translated operands.

² In the right-hand side, juxtaposition is the right-associative application of a funcon to a (single) funcon term, i.e. $\mathbf{bound\ string\ } lit == \mathbf{bound(string(lit))}$.

3.3.3 Operator Specialization

In certain circumstances, it may be necessary to adapt the semantics of language constructs in order to make them suitable for the language in mind. The so-called ‘glue code’, which adapts an existing semantic definition, is often used in these circumstances. This glue code is to be written modularly and in isolation, without anticipating, or constraining, future interactions. These observations can be exemplified by the following example: Consider an *if* operator encoding if-expressions or if-statements.

$$\begin{aligned} If_{\mathcal{O}} &: IfCond \times IfTrue \times IfFalse \\ If_{\mathcal{F}}(c, t, f) &= \mathbf{if\text{-}true\text{-}else}(c, t, f) \end{aligned}$$

The **if-then-else** funcon expects that the conditional evaluates to a boolean. However, in C-like languages, if-statements are defined in terms of integers. Therefore, to utilize $If_{\mathcal{O}}$ we might glue it into our C-like language as shown below.³

$$\begin{aligned} CExpr \subseteq IfCond & \quad (\text{Sort constraint with glue code}) \\ \hookrightarrow \mathbf{not\ is\ equal}(0, CExpr_{\mathcal{F}}) & \quad (\text{glue code}) \end{aligned}$$

In the example, we perform a sub-sort declaration, linking the $CExpr$ sort — containing the C expression operators — to the $IfCond$ operand. As part of that declaration, we define glue code which is only applied when the translated operator is part of the $CExpr$ group, since glue code is conditional. Furthermore, the glue code glues the result of the translation function of the operator on which glue code is defined. Thus in our example, $CExpr_{\mathcal{F}}$ is the result of the translation function associated with the sort $CExpr$, which is implicitly defined in terms of the translation functions given for the operators contained in the sort $CExpr$, i.e. the catamorphism.

We thus have specialized the *If* operator to the semantics of our specific language without modifying the existing definition of the *If* operator nor do we need to define a different operator for all possible variations. By applying glue-code conditionally,

³ The example is simplified to save space. When performing such glue on C, the checks need to be extended to supports floats, doubles, etc. This can be easily done by dispatching on the type of the current value.

other operators assigned to *IfCond* are not affected, and removing C-like expressions from the language does not leave any stale glue code around.

3.3.4 Language Definitions

Given a set O of operators, with every operator having an arity, denoted with $|o|$, a set of operand positions, denoted with $\vec{o} = \{1, \dots, |o|\}$, and a semantic function $F(o) : \mathcal{F}^{|\vec{o}|} \rightarrow \mathcal{F}$ where \mathcal{F} is the set of function terms, we define a language as a structure $\langle T, S, G, I \rangle_O$ in terms of O , with $T \subseteq O$ being the set of top-level operators; S is a family $\langle S_{o \in O, w \in \vec{o}} \rangle$ of sets indexed by $O \times \mathbb{N}$. $S_{o,w}$ is the set of operators assigned to the operand position w of operand o . G is a family $\langle G_{o \in O, w \in \vec{o}} \rangle$ of functions indexed by $O \times \mathbb{N}$. $G_{o,w}$ is the glue function $O \times \mathcal{F} \rightarrow \mathcal{F}$ for operators assigned to the operand position w of operand o . I is a family $\langle I_t \in T \rangle$ of functions indexed by the top-level operators. $I_t : \mathcal{F} \rightarrow \mathcal{F}$ denotes the top-level initialization function for the specific top-level operator.

In the mathematical formulation we do not distinguish between sub-sort declarations and operator assignments, since all sub-sort declarations can be described in terms of operator assignments. Furthermore, the definition does not introduce operand names. Instead, operand positions are used. Nevertheless, we do use names in our examples as a notation convenience, ensuring that there is a one-to-one mapping between operand names and operand positions.

The top-level operators are present in a language to determine the entry points of the language. This can be used in the generation of parsers for languages, generation of tooling, generation of language structure diagrams, etc. Initialization functions can be used to modify the top-level behavior of the language. For example, in a REPL, returned values might be printed and unhandled exceptions caught and displayed while not resulting in termination of the REPL. This behavior is however not preferred when not running in a REPL-like environment. With the initialization functions, this behavior can be encoded as an extension on a language, creating a clear distinction between the tools as languages.

3.3.5 Language composition

Language composition (\diamond) of two languages, L_1 and L_2 , specified in terms of the same operator set, is defined as follows.

Definition 3.3.1. $L_1 \diamond_O L_2 = \langle T_1 \cup T_2, S, G, I \rangle_O$, where

$$\begin{aligned} S &= \{S_{1\langle o,w \rangle} \cup S_{2\langle o,w \rangle} \mid o \in O, w \in \vec{\sigma}\} \\ G &= \{G_{2\langle o,w \rangle} \circ G_{1\langle o,w \rangle} \mid o \in O, w \in \vec{\sigma}\} \\ I &= \{I_{2\langle t \rangle} \circ I_{1\langle t \rangle} \mid t \in T_1 \cup T_2\} \end{aligned}$$

From the associativity of the operations used on the elements of the languages, it follows that language composition is associative. Language composition, however, is not commutative due to the usage of function composition with G and I . With language composition, languages form a monoid. The neutral language can be defined by taking the empty set for T , letting the family S assign the empty set to every index, and letting the families G and I assign the identity function over function terms to every index.

Defining languages in terms of a fixed operator set does not restrict the incrementality we provide nor does it prevent new operators from being introduced. New operators can be added at any time, because operators and their semantics are compositional as well, as inherited from the conceptual model of data types à la carte and our usage of initial algebra semantics. The compositional nature provides enough to support the incremental aspect. In essence, when composition is supported, incrementality is almost obtained for free, because we can always reformulate an incremental step as a composition from our starting point. Since every incremental step is then evaluated as a composition from the starting point, we do assume that the interpretation of this composition scales and is not much slower than just doing the incremental step. This idea can be visually explained in Figure 3.1.

Incrementality in the conventional way is obtained by executing small programs in isolation and keeping track of the current configuration. Our approach, instead, keeps track of the initial configuration and all submitted programs, an evaluation then always starts from the initial configuration.

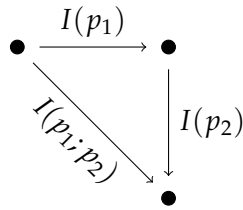


Figure 3.1: The idea of incrementality via composition shown visually. Interpreting (I) the first program (p_1) and then the second (p_2) gives the same result as interpreting the sequence of p_1 and p_2 ($p_1; p_2$). We can achieve incrementality by always evaluating the composition from the initial state.

iCoLa thus exists out of two languages: one for defining operators and their semantics, and one for defining languages. The interpretation of the operator language results in a set of operators that is used in the interpretation of the language-definition language. Because both operators and languages are compositional, from a users perspective there is no difference, and language and operator definitions can be freely mixed. This, again, is a result of our approach of achieving incrementality via composition.

3.4 IMPLEMENTATION

In this section we demonstrate an EDSL in Haskell implementing the approach introduced in the previous section. The EDSL is partly embedded in Haskell and partly embedded in Template Haskell. In case definitions in the EDSL must be given in terms of Template Haskell, we first give the Haskell definition, followed by the corresponding encoding in terms of Template Haskell. This demonstrates to which Haskell expression the Template Haskell encoding is evaluated by our approach.

3.4.1 Operators

Operators are implemented as GADTs with two type parameters, u and t , corresponding to the universe (set) of operators, which is needed for injection into the co-product type, and a so-called meta-type (explained below) of the operator, respectively.

GADTs are needed to support the delayed decision regarding sort-constraints via class instances in the constructor definition of the operator. To illustrate, we take the definition for $Abs_{\mathcal{O}}$ from §3.3 as an example.

```

data Abs u t where
  Abs :: IsTrue (AbsBody t) 1a =>
    String → u t 1b → Abs u AbsType 2a
type family AbsBody 1c t
data AbsType 2b

```

The sort-constraint (1a) enforces that the second parameter (1b) is assigned to the `AbsBody` sort, and sorts are implemented as type-families (1c). Because we carry around the meta-type in arguments (1b) to enforce sort-constraints, type `u` is a type taking a type as a parameter, i.e., it is a functor. Consequently, our operators are higher-order functors. In addition, every operand is assigned a meta-type (2a), which are implemented as proxy types (2b) — data types without constructors. Furthermore, `IsTrue` is a type class for which only one instance is defined: the instance for the type-level boolean `True`. This enables boolean programming at the type level.

```

class IsTrue bool
instance IsTrue True
data True

```

To encode $Abs_{\mathcal{O}} \in AbsBody$ — the assignment of the abstraction operator to the body of abstractions — we define the meta-type of the operator as an instance evaluating to `True` of the `AbsBody` type family.

```

type instance AbsBody AbsType = True

```

To retain adaptability of operator assignments, we delay the instantiation of such instances by defining them in terms of Template Haskell.

```

('AbsType, 'AbsBody) :: OperatorAssignment
type OperatorAssignment = (MetaType, Sort)
type MetaType = Name
type Sort = Name

```

Auxiliary sorts are also implemented using type families. For example, our earlier convenience sort, `Expr`, is defined as follows.

type family Expr t

As such, instances can be added to an auxiliary sort with an operator assignment.

To perform sub-sort assignments, a sort is linked to another sort, again in terms of template Haskell. Thus, the encoding for $Expr \in AbsBody$ is as follows.

```
('Expr, 'AbsBody) :: SubSort
type SubSort = (Sort, Sort)
```

3.4.2 Semantic Functions

As is the case in data types à la carte, semantic functions are defined modularly as type class instances and are applied by the fold of an algebra. For example, the following instance encodes the definition of $Abs_{\mathcal{F}}(x, b)$ given in §3.3.2.

```
instance ToFuncons Abs where4
  toFuncons (Abs s (K body)) = K $ function_ [closure_
    [scope_ [bind_ [T.string_ s, given_], body]]]
```

The ToFuncons type class captures those types for which a translation to ‘Funcons’ is available and is defined as follows.

```
class HFunctor f => ToFuncons f where5
  toFuncons :: Alg f (K Funcons)
newtype K a i = K {unK :: a} deriving (Functor)
type Alg f e = f e :-> e
type (:->) f g = forall i. f i -> g i
```

Since operators are higher-order functors and we carry around the meta-type, the carrier of our algebra must be a parameterized functor, which funcons are not. Therefore, we wrap funcons with the K constructor. The K constructor simply wraps a value, but adds an additional component at the type level, giving the type K the kind $\star \rightarrow \star \rightarrow \star$, corresponding to a higher-order functor.

⁴ The T module provides helper functions to transform Haskell values into funcon values. Funcon smart constructors — identified by the trailing underscore — take a variable number of arguments, hence the usage of lists.

⁵ The K, Alg and (:->) . types are defined by the comp-data library [14].

3.4.3 *Glue Code*

Glue code is implemented as endofunctions on the `funcons` type, and to link glue code to an operand of a specific operator, we use multi-parameter type classes and an adaptation on the `cata` morphism that applies glue code before the application of the semantic function on the operator being evaluated. For example, the glue definition from §3.3 is achieved with the following instance definition.

```
instance GetGlue IfGlue CExpr Funcons where
  getGlue IfCondGlue _ = \l → not_ [is_equal_ [0, l]]
  getGlue _ _ = id
class GetGlue operand (f :: (* → *) → * → *) target where
  getGlue :: operand → f (Term a) b → target → target
  getGlue _ _ = id
```

In this example, `CExpr` refers to all operators assigned to the `CExpr` sort, and should be read as the generation of this instance for all operators assigned to the `CExpr` sort.

To identify the operands, we generate a data type for every operator, `IfGlue` in the example, where the cases identify the operand positions for the application of glue code, which in the example are identified by the constructors suffixed with `Glue`.

Again, to be able to modify glue code instances, we delay such instances by defining them in terms of Template Haskell constructs instead.

```
('CExpr, 'IfCond, 'cExprGlue) :: GlueDefinition
type GlueDefinition = (MetaType, Sort, GlueFunction)
type GlueFunction = Name
cExprGlue :: Funcons → Funcons
cExprGlue l = not_ [is_equal_ [0, l]]
```

The Template Haskell definition does not refer to the glue data type. Instead, the right glue data-type is automatically determined based on the sort in the glue code definition. The glue data-type is thus fully abstracted away from language designers.

3.4.4 *Language Definitions*

Because we defined the components of a language in terms of template Haskell constructs, we can define languages themselves as data types in terms of those template Haskell constructs.

```

data Language = Language
  { op_assign :: [OperatorAssignment]
  , sub_sorts :: [SubSort]
  , glue_code :: [GlueDefinition]
  , init_code :: [(MetaType, GlueFunction)]
  }
instance Semigroup Language
instance Monoid Language

```

The language definition does not contain a special entry for top-level operators. Instead, we utilize a special sort — `TopLevel` — that can be used inside the operator assignment and sub-sort definitions. As a result, initialization code is defined as a tuple, linking operator — via their meta-type — to glue-code functions. In addition, we make languages an instance of the `Semigroup` type class, allowing usage of the (`<>`) operator to compose languages.

To instantiate a language, we define the `genLanguage` function that can be spliced in.

```

genLanguage :: [Operator] → Language → Q [Dec]
type Operator = (Constructor, MetaType)

```

The first argument denotes the operator set in which the second argument — the language — is defined in. This first argument is needed for the generation of glue code and parsers. The `genLanguage` function distributes the sub-sort declarations over the operands, generates type family instances for the operator assignments, generates smart constructors that automatically inject the operator into the co-product type (representing the set of operators), generates glue code data types for the operators, and the glue code definitions are transformed into `GetGlue` type class instances. Furthermore, the function performs several checks. First, it checks if all operators mentioned in the language are present in the set of operators. Secondly, it requires that the sub-sort declarations form a directed acyclic graph. If any of these conditions are not met, the compilation is stopped with an error indicating the unsatisfied condition.

Our original definition of the lambda calculus is thus obtained by the definition in Listing 3.2.

Listing 3.2: Definition of the lambda calculus in *iCoLa*.

```
$(genLanguage [('Var', 'VarType'), ('Abs', 'AbsType'),
              ('App', 'AppType')] lambdaLanguage
  where
    lambdaLanguage = Language
      { op_assign = [(op, 'Expr') | op ← ['VarType', 'AbsType', 'AppType]]
      , sub_sorts = [('Expr', t) | t ← ['AbsBody', 'AppLeft', 'AppRight', 'TopLevel']]
      , glue_code = []
      , init_code = []
    }
```

3.4.5 Parser Generation and the *iCoLa*-shell

The structure present in language definitions is enough to use in the generation of parsers for the defined languages. Currently, we generate a parser that parses a language in which operator application is written in a style similar to LISP [128]. We use such syntax to ensure that the generated grammar is not ambiguous. For example, `(add (intv 1) (add (intv 2) (intv 3)))` demonstrates an expression using the generated syntax for a simple integer addition language. In this example, `add` and `intv` are operators, and applications of operators are always surrounded by parentheses. The operands are separated by spaces. So this example simply encodes the arithmetic expression $1 + (2 + 3)$.

Using these generated parsers, we provide the *iCoLa*-shell, which is a ‘meta-REPL’ that accepts any Haskell declaration. This enables users to define languages and operations over languages inside the meta-REPL. Furthermore, a meta-command `commit` is provided that can be used to commit to a specific language and start an ‘object-REPL’ for the chosen object language. In the object-REPL, the user can experiment with the defined language using the LISP-style generated concrete syntax. When stopping the object-REPL, the user returns to the meta-REPL, continuing the same session where they left off. Listing 3.3 illustrates an example session in the meta-REPL and an object-REPL for a simple integer addition language, where `meta>` denotes execution

inside the meta-REPL, otherwise execution is happening inside the object-REPL.

Listing 3.3: Example session inside the *iCoLa*-shell.

```
meta> intAdd = intLanguage <> addLanguage
meta> :commit intAdd
intAdd> (add (intv 1) (intv 10))
11
intAdd> :exit
meta>
```

In the same *iCoLa*-shell session, the user can extend the language and start a new object-REPL, or focus on a subset of the language by removing part of the composition or focusing on a subset of languages used in the composition.

3.5 EXAMPLES OF INCREMENTAL COMPOSITIONAL LANGUAGE DEFINITIONS

In this section we showcase *iCoLa* by defining several demonstration languages in terms of other languages. The used languages are: *Imp* [181], a simple imperative language; *SIMPLE* [182], a more complex procedural language; and *MiniJava* [8], a strict subset of the Java language. These languages are chosen because they have their semantics described in terms of funcons as part of the case studies for the PlanCompS project.⁶ This enables us to focus on the incremental and flexibility aspects of our approach and to demonstrate the reuse achieved via operator definitions and glue code.

3.5.1 *The Construction of Imp*

We define *Imp* as the composition ($\langle \rangle$) of the following four languages: *impArith* $\langle \rangle$ *impBExpr* $\langle \rangle$ *impStmts* $\langle \rangle$ *impPrograms*, a simple arithmetic language with support for integer addition and division; a boolean expression language with support for less-than-equal comparison and (binary) conjunction; a statements language containing if-statements, while-statements, and sequencing of statements; and a program language that unifies

⁶ <https://plancomps.github.io/CBS-beta/docs/Languages-beta/index.html>

these languages together by defining the top-level in accordance to the top-level of *Imp*, respectively. The definition of `impArith` and `impBExpr` are as follows.

```
impArith = Language
  { op_assign = [(e, 'ArithExpr) |
    e ← ['IntType, 'AddType, 'DivType, 'IdType]]
  , sub_sorts = [('ArithExpr, s) |
    s ← ['AddLeft, 'AddRight, 'DivLeft, 'DivRight]]
  , glue_code = [('DivType, 'Always, 'check_divide)]
  ...} where
  check_divide f = checked_ [f]

impBExpr = Language
  { op_assign = [(e, 'BExpr) |
    e ← ['BoolType, 'LeqType, 'NotType] ]
  , sub_sorts = [('BExpr, s) |
    s ← ['NotExpr, 'AndLeft, 'AndRight]]
    ++ [('ArithExpr, s) | s ← ['LeqLeft, 'LeqRight]]
  ...}
```

In these definitions, there are four things that need to be highlighted. Firstly, both languages do not define a top-level, which means that these languages on themselves are not executable since there are no entry points. Secondly, `impArith` defines glue code over the *Div* operator that wraps the divide in a check. When division by zero occurs, the program is terminated due to the check. This behavior is not directly encoded in the semantics of the *Div* operator, because other languages handle this differently, for example by throwing an exception. Thirdly, in the description for the glue code we see usage of the shorthand "Always sort. This sort is a convenience and denotes that the glue code needs to always be applied on the *Div* operator, it thus encodes the assignment of this glue code to all operands to which *Div* is assigned. Finally, `impBExpr` uses the *ArithExpr* auxiliary sort in its definitions but does not assign any operators to this sort. Thus, `impBExpr` is an extension on the arithmetic language.

Alternatively, we can define a refined version of `impBExpr` that is independent from `impArith` as follows.

```
impBExpr- = Language
  { op_assign = [(e, 'BExpr) |
    e ← ['BoolType, 'LeqType, 'NotType]]
  , sub_sorts = [('BExpr, s) |
    s ← ['NotExpr, 'AndLeft, 'AndRight]] ...}
```

In this definition, we removed the mentioning of the auxiliary sort. As a result, the refinement of `impBExpr` is not an extension. To get back to our original definition of `impBExpr`, we can define a new language that glues `impArith` and the refined version of `impBExpr` together. This glue language only contains the sub-sort declaration that we removed, modeling language unification.

```
impArith <> impBExpr- <> glueLanguage
  where
    glueLanguage = Language
      { op_assign = []
      , sub_sorts = [('ArithExpr', s)
                    | s <- ['LeqLeft', 'LeqRight']]
      ... }
```

Alternatively, owing to our languages being first-class citizens in Haskell, we can define `impBExpr` as a function with one parameter denoting the sort that can occur in less-than-equal expressions.

```
impBExpr+ leqSort = Language
  { op_assign = [(e, 'BExpr') |
                 e <- ['BoolType', 'LeqType', 'NotType']]
  , sub_sorts = [('BExpr', s) |
                 s <- ['NotExpr', 'AndLeft', 'AndRight']]
  ++ [(leqSort, s) | s <- ['LeqLeft', 'LeqRight']]
  ... }
```

This makes the parameterized version of `impBExpr` configurable and removes the hard dependency on the auxiliary sort while also removing the requirement of a glue language. Instead, we can decide the correct auxiliary sort when composition occurs.

3.5.2 Reusing *Imp* to Define *SIMPLE* and *MiniJava*

We can reuse the definition of *Imp* to define *SIMPLE* and *MiniJava*, utilizing *Imp* in different ways. *SIMPLE* is a much more elaborate language that almost fully subsumes *Imp*. *MiniJava* is less elaborate since it does not contain all constructs present in *Imp*, but still shares a significant part, some of which is showcased in Table 3.1. However, the way *Imp* is defined does not fully correspond with both *SIMPLE* and *MiniJava*, since the top-level of *Imp* is different and *Imp* contains certain constructs not present in *MiniJava*. To align *Imp* with these requirements we refine the language to a

Listing 3.4: An example showing two refinements, one removing top-level operators, and one removing the less-than-equal operator from the language.

```

 $\psi$  lang = lang { sub_sorts = removeTopLevels . sub_sorts $ lang }
  where
    removeTopLevels = filter $ not . (=='TopLevel') . snd

 $\phi$  lang = lang { op_assign = removeLeq . op_assign $ lang
  , sub_sorts = removeLeqOps . sub_sorts $ lang }
  where
    removeLeq = filter $ not . (=='LeqType') . fst
    removeLeqOps = filter $ not .
      (flip elem ['LeqLeft', 'LeqRight']) . snd

```

language that aligns with both *SIMPLE* and *MiniJava*. A refinement is an endofunction over languages implemented directly as a Haskell function. This way, all components of a language can be refined, closely resembling the idea of restriction as presented by Erdweg [65].

To align *Imp*, we define two refinement functions in Listing 3.4, one, denoted with ψ , that removes the operators from the top-level as defined by *Imp* and one, denoted with ϕ , removing the less-than-equal operator from the language. For *SIMPLE*, ψ is enough to make *Imp* suitable to use in the definition. In case of *MiniJava*, both ψ and ϕ are needed, thus the composition of these refinement functions is the required refinement function for *MiniJava*.

Besides the required refinement, *Imp* makes a distinction between two types of expressions: arithmetic expressions and boolean expressions. Variables can only occur inside arithmetic expressions and not in boolean expression. In the definitions of *SIMPLE* and *MiniJava* this distinction is not made. Nevertheless, because we make a distinction between operators and sorts in our approach, these structure choices do not prevent the usage of *Imp* when defining both *SIMPLE* and *MiniJava*, because we can define a new auxiliary sort and link both *Imp* sorts to this new sort and then distribute the new sort over the required operands. This demonstrates the flexibility of our approach and that existing

Table 3.1: Table highlighting some operators used in the definition of *Imp*, *MiniJava*, and *SIMPLE*. The rows indicate operators used during the evaluation and the columns the constructed languages from the collection. The ● indicates that the operator is used as is; ◐ indicates that an operator is used with glue code; and ○ indicates that an operator is not used.

	<i>Imp</i>	<i>MiniJava</i>	<i>SIMPLE</i>
Int + Addition	●	●	●
Substraction + Multiplication	○	●	●
Division	◐	◐	◐
If + While	●	●	●
Variables	●	●	●
Ouput	○	◐	●
Input	○	○	●
Classes	○	●	○
Arrays	○	◐	◐
Throw + Catch	○	○	●

language structure choices do not restrict in which compositions a language can be used.

Both *SIMPLE* and *MiniJava* also add new constructs that are not present in *Imp*. Some of the constructs occur in both *SIMPLE* and *MiniJava*. Table 3.1 highlights some of the language constructs used and their presence in the languages. This table is not extensive and we sometimes group operators together due to space limitations, since *SIMPLE* alone already contains 40 language constructs. Nevertheless, it demonstrates a selection of constructs that are often present in multiple of the defined languages. This highlights the reusability obtained via our approach.

While operators might occur in multiple languages, their usage is not always identical. For example, in *MiniJava*, output is always followed by a newline, which is not the case in *SIMPLE*. Also, both languages check out-of-bounds array access, hence the required glue code.

Table 3.2: Table demonstrating the construction of several languages with a fixed-set of operators. Columns indicate the operators used during the evaluation and the rows are the languages constructed with (some) operators from the collection. The ● indicates that the operator is used as is; ◐ indicates that an operator is used with glue code; and ○ indicates that an operator is not used.

	Var	Abs	App _{cbv}	Add	Int	Return	Call/cc	If	Throw	Catch
lambda	●	●	●	○	○	○	○	○	○	○
arithmetic	○	○	○	●	●	○	○	○	○	○
exceptions	○	○	○	○	○	○	○	○	●	●
proc	●	◐	●	○	○	●	○	○	○	○
lambda _{cbn}	◐	●	◐	○	○	○	○	○	○	○
functional	●	●	●	●	●	○	●	●	●	○
procedural	●	◐	●	●	●	●	○	●	●	●
procedural + functional	●	◐	●	●	●	●	●	●	●	●

3.5.3 Object Language Variability

MiniJava is interesting because variations of *MiniJava* exist that have been introduced for teaching purposes.⁷ A teacher can adapt the experience of students based on their expertise using language variants or by growing the language throughout the course [39, 84]. This flexibility is naturally supported by our system since the (full) *MiniJava* language can be given as the composition of multiple smaller language variants. This enables a teacher to exclude or include languages to create new variants. Furthermore, a teacher is not restricted to the existing core of *MiniJava*, because with sort-constraints, a teacher can freely alter the language to their needs. For instance, a teacher can remove the object-oriented aspect of *MiniJava* and start with procedural programming before introducing objects and classes. Alternatively, a teacher can include the Exceptions from *SIMPLE* to add exceptions to *MiniJava*.

⁷ <https://courses.cs.washington.edu/courses/cse401/13wi/project/MiniJava.html>; <http://teaching.up.edu/cs358/miniJava.pdf>

Language variability is also useful when designing a programming language. In the *iCoLa*-shell, different variants of a language can be defined and tested with relative ease. Multiple variants can exist side-by-side, making it easy to compare and contrast variations and gather early feedback to include in the design process. In Table 3.2, the outcome of such a session is listed as a table. In this session, a fixed set of operators is used to define a variety of languages. Language definitions were defined in isolation or via composition. For instance, *lambda_{cbn}* is defined by composing the *lambda* language with a language consisting (only) of glue-code that inserts the semantics of call-by-name using thunks.

```
lambdaCBN = lambda <> Language
{glue_code = [('Always', 'AppArg', 'thunk'),
              ('VarType', 'Always', 'force')]
...} where
  thunk f = thunk_ [f]
  force f = force_ [f]
```

In this definition, we assume that all variables are assigned to thunked values. This is not always the case, e.g. in a procedural language with global variables. Type information can be used to distinguish variables based on whether their values are thunked. This, however, is not possible in our glue code definitions because glue code is context-free. However, it can be realized within the semantic domain of funcons, as funcon terms are dynamically typed. The table shows an overlap between different languages and the two forms of variability in our approach: we can add new operators to existing languages and add new languages using existing operators, without modification of existing code.

3.6 DISCUSSION

In *iCoLa*, some of the techniques discussed in §3.2 are combined as follows. The syntax and semantic functions of operators are defined modularly using data types à la carte. The funcons of Funcons-beta are used to express the semantics of all operators in the same semantic domain. This makes it possible to define languages by selecting (top-level) operators from the set of all available operators. This is consistent with the methodology of [31]

and ensures object languages in *iCoLa* are ‘sequential languages’ by definition. As such, REPLs for the object languages are obtained for free. The meta-language is sequential in itself, thereby supporting incremental meta-programming in the *iCoLa*-shell. This is achieved by defining operators in isolation using ‘sort constraints’ as explained in §3.3. The sort constraints are enforced statically by applying (Template) Haskell in the implementation of *iCoLa* as an EDSL.

In this section we discuss the effect of our choice to embed *iCoLa* inside (Template) Haskell. We look at restrictions in the current implementation, such as the absence of user-defined concrete syntax, that constrain the usage of *iCoLa*. And discuss the scalability of our choice to use sort constraints instead of conventional abstract syntax definitions.

3.6.1 *iCoLa* as an EDSL

The presented implementation is in the form of a Haskell EDSL. The EDSL offers static guarantees such that every operator in a language has an associated semantic function and sort constraints are respected. In addition, language designers have the full power of Haskell available to them when defining and manipulating languages. However, we are also restricted by our choice of implementation. Because we utilize Template Haskell, a compilation step is needed before a defined language can be used, only one language can be generated per module, and there is a stage restriction enforced by Template Haskell. This requires us to implement the *iCoLa*-shell separately instead of reusing the REPL provided by the Haskell compiler (e.g. GHCi), and makes the feedback loop more convoluted. Furthermore, we assume Haskell familiarity from language designers, for example to understand Haskell type errors when operators are incorrectly used. In addition, operators, semantic functions, and language definitions involve some boilerplate code. For example, the introduction of type families for operators, the need for constraints on operator definitions, and occurrences of the `K` constructor.

Some boilerplate code can be removed by using quasi-quotation. With quasi-quotation, we could provide a small layer of syntactic sugar that removes most of the boilerplate code currently present

in our approach. The concern regarding Haskell type-errors can be mitigated by providing custom type-errors or other strategies for type-error customization in EDSLs [195].

Another alternative is an implementation with an external DSL. This gives the possibility to provide syntax that is much closer to the mathematical notation of §3.3 and also allows us to give more domain-specific error messages. However, such an implementation is more complex and puts a restriction on the semantic functions. Currently, our implementation is easily extended with new semantic functions; and because semantic functions are implemented in Haskell, the possibilities are endless. When providing an external DSL, this flexibility is lost, requiring either a constraint on the semantic functions or complicating the implementation to support more complex semantic functions. Using an external DSL also removes many of the benefits we currently have by providing languages as first-class citizens inside Haskell. Alternatively, a hybrid approach can be implemented that provides a DSL layer on top that compiles down to Haskell such that semantic functions and language refinement can still be defined as Haskell functions. How these different implementation techniques affect usability and expressiveness is to be explored in future work.

3.6.2 *Restrictions and Scalability*

With the flexibility our approach provides, language definitions can become unwieldy where it is unclear where operators are exactly assigned to, which operators are part of the language, and how they are affected by glue code. In our experience from our case studies, the development is often done in steps, where prototyping is modularized by making local modifications such that the effects of the modifications are easier to observe. However, this is not always possible. In future work, we want to explore tooling that can help in quickly understanding the effects of new operator assignments and language composition. For example, via the structure of the language, a BNF-like grammar can be generated that shows the structure in an uncluttered fashion. In addition, we envision tooling that highlights the effects of language composition and allows one to zoom in on specific

operators and see how they are affected by glue code. Furthermore, the algebra used in our approach has no context, which requires that the translation must be done in a context independent manner. This defers a lot of the work to the semantic domain, but keeps operator implementations simple. In future work, we would like to explore having additional algebras defined as semantic functions, especially to express static semantics, without losing the flexible compositional capabilities of our approach. We set a first step into this direction in the next chapter.

3.6.3 *Concrete Syntax*

In the presented framework, a LISP-style concrete syntax definition is generated for object languages. However, a language designer might want to define their own concrete syntax. To support user-defined syntax, we can extend operator definitions with a notion of concrete syntax or extend language definitions to attach concrete syntax to the chosen operators of the language. In both cases, problems with ambiguity can arise when combining concrete syntax definitions. Generalized parsing techniques, such as Early parsing [62], GLR [202], and GLL [192], can be used to accept ambiguous grammars. Handling ambiguities without much effort is important, as even the introduction of a single operator into a language can result in ambiguity, hampering rapid prototyping when ambiguities must be resolved. When a language is finalized, the grammar can be inspected and adapted to possibly remove ambiguities.

An alternative approach is to mix user-defined parsers and generated parsers. This way, operators are not extended with a notion of concrete syntax. Instead, a language designer develops a parser alongside the language definition and when prototyping with new operators uses the generated parser for those operators. The parser is then incrementally defined just like the language itself. However, this approach requires an efficient way to connect user-defined parsers and generated parsers, and further integration to ensure the language development process can fully occur inside the *iCoLa*-shell. Nevertheless, in both cases the existing implementation of parser generation can be largely reused, since the generation of operand parsers is generic.

In the next chapter we investigate an approach to incorporate user-defined concrete syntax into our current model, and evaluate the effect of this extensions on the interactive, reusable, and compositional aspects of our approach.

3.7 RELATED WORK

Developing languages via some form of composition is supported by a wide variety of language-development environments [66, 96, 201, 204, 213]. Erdweg et al. [65], performed a systematic evaluation of existing environments and their support for the different forms composition (extension and unification). Out of the considered environments, only JastAdd [63], which is an environment for the construction of Java like languages, supported unification at the semantic level. Nevertheless, extension-unification is supported by most environments.

Lisa [135] is a full-fledged interactive environment for programming language development based on attribute grammars with support for incremental language development [136] via multiple attribute grammar inheritance [134]. Compared to our approach, the incremental focus is more linear and distinction between operators definitions and usages is not made.

Melange [59] is a meta-language involving meta-models and aspect oriented programming. It uses aspects to implement the semantics of languages, and supports both extension and unification. Our operator specialization closely resembles the idea of aspects as seen in Melange. In contrast to our approach, Melange makes no distinction between operator semantics and operator specialization, possibly reducing reuse opportunities. For abstract syntax definitions (operator definitions), the approach uses Ecore,⁸ which is more rigid than our operator definitions but does provide a significant reuse opportunity since many Ecore metamodels exist.

In Feature-oriented programming [7], a system is decomposed in the features it provides. This style of programming aims to increase structure, reuse and variation by making features user configurable such that a system can be developed by picking

⁸ <https://www.eclipse.org/modeling/emf/>

and configuring the correct features. Neverlang [38] is a development environment modeled around the idea of feature-oriented programming, where features are implemented using an object-oriented approach.

Software product lines [43] is a development paradigm that models the software development process as a product line, where a system is constructed by selecting components from a repository, somewhat resembling our idea of an operator universe from the language point of view, adapting the components to the use case, and integrating the components together. Compared to feature-oriented programming, software product lines focus on similarities between systems, also known as family systems. This gives a high variability where variants of systems can be quickly created. Feature-oriented programming can be used to implement software product lines, which is done by AiDE [110]. AiDE provides an environment for language-development based on software product lines by building an environment on top of Neverlang [38]. Besides AiDE, there are several other environments integrating software product lines in the context of language development — also known as language product lines [131].

Focus on language families [123], a set of related languages, is inherent in the language product lines style of development. As a result, the variability of these systems is high enabling the construction of a wide variety of languages in an incremental manner. However, because the focus is on language families, there is a restriction on the structure of the different variations.

Solutions to the expression problem, such as finally tagless [37], object algebras [186], and, data types à la carte [199], naturally lead to an extensible approach for operators and can be used to implement languages in a modular fashion, as demonstrated by several systems based on the solutions to the expression problem [78, 87]. However, composition and variability are not necessarily obtained. Nevertheless, it would be interesting to see if these solutions to the expression problem can function as a compilation target, for which we use data types à la carte now, in our implementation.

LANG-N-PLAY [42] is a proof-of-concept language introducing the idea of *languages as first-class citizens*. *LANG-N-PLAY* is a statically typed functional language in which languages are

just expressions. Consequently, operators over languages are defined as functions. In contrast to our approach, *LANG-N-PLAY* is a newly developed (experimental) language. Our approach is embedded in a general-purpose language, albeit via template programming. The usage of Template Haskell can complicate type errors reported to the user, since the Template Haskell expression is not type checked, but the resulting program is [197]. *LANG-N-PLAY* is also statically typed but without the intermediate template layer.

3.8 CONCLUSION

In this chapter we introduced *iCoLa*, a meta-language aimed at improving the language design process through rapid prototyping with reusable components and incremental programming. The *iCoLa*-shell enables fast prototyping by supporting the simultaneous definition of multiple languages that can be composed, unified, extended, restricted and tested within a shared REPL session. In *iCoLa*, languages are first-class citizens such that the users are given the full power of the host language (Haskell) to define languages. Operators over languages can be defined (e.g. composition) and languages can be parameterized, including by other languages (e.g. refinement). By constructing several languages with our approach, we have demonstrated to which extent our approach simplifies the construction of new languages as well as variants of existing languages.

The flexibility provided by *iCoLa* makes it difficult to track the precise composition of a language when applied at (large) scale and user-defined concrete syntax is currently not supported. Methods to improve *iCoLa* in these regards are explored in Chapter 4.

The previous chapter introduced *iCoLa*, implemented as an EDSL in Haskell. In this chapter we extend this work by introducing *iCoLa*⁺, which extends the formal model with support for an arbitrary amount of domains and concrete syntax definitions for operators, and an implementation of this model as an external DSL. The implementation of *iCoLa*⁺ enjoys three interaction levels, enabling extension points for different domain experts, namely language engineering experts and user interaction experts, to extend some of the capabilities of *iCoLa*⁺. To demonstrate *iCoLa*⁺, we perform an extended replication of the evaluation performed on *iCoLa* (in the previous chapter). Additionally, we perform a small exploration of defining DSLs in *iCoLa*⁺ by looking at a variant for the DOT language.

Associated Publications

- **Damian Frölich** and L. Thomas van Binsbergen. “iCoLa+: An extensible meta-language with support for exploratory language development.” In: *Journal of Systems and Software* 211 (2024), p. 111979. ISSN: 0164-1212. DOI: <https://doi.org/10.1016/j.jss.2024.111979>

4.1 INTRODUCTION

Since this chapter is an extension of the work presented in the previous chapter, the high-level motivation is unchanged. Instead of repeating this, we highlight the motivation to develop an external DSL implementation, and the motivation to extend to the formal model with support for an arbitrary amount of semantic domains and concrete syntax definitions.

A primary goal in this thesis is to support exploratory programming at the meta-level. With *iCoLa*, we obtain support for

exploratory programming for the defined object-language, but obtaining it for *iCoLa* itself is difficult due to the dependency on Template Haskell. Furthermore, *iCoLa* does not support user-defined concrete syntax for operators. This can significantly hamper the experience, since having to context-switch to define parsers is not ideal, and syntax can affect the experience of using a language. Although the implementation of *iCoLa* does not restrict the used semantic domains owing to the choice of using data types à la carte, the formal model does not take this into account and is only defined with funcons as the semantic domain.

Motivated by these observations, we introduce *iCoLa*⁺, an extensible meta-language with a focus on the language design process via exploratory language development and rapid prototyping, achieved by utilizing reusable language components and supporting exploratory programming at both the object and meta-level. *iCoLa*⁺ builds upon *iCoLa* by providing an extensible implementation that supports user-defined environment and DSL-based domain definitions, as well as extending the approach with concrete syntax, and also supporting an arbitrary amount of semantic domains. Specifically, we make the following contributions:

- We extend the approach of *iCoLa* (presented in Chapter 3) with concrete syntax and support for an arbitrary amount of semantic domains (§4.3).
- We provide an extensible implementation of the extended model in the form of a DSL. The implementation also functions as an alternative to the implementation presented in Chapter 3 (§4.4).
- We evaluate the extended approach and compare the new implementation with the *iCoLa* implementation (§4.5).

This chapter is outlined as follows: in §4.2 we give the required background. The formal model of *iCoLa* is extended with concrete syntax definitions and support for an arbitrary amount of semantic domains in §4.3. In §4.4, a DSL for the extended model and its implementation in Haskell is presented. The extended approach is evaluated via an extended exact replication in §4.5. A

discussion on the two implementations and the extended model is held in §4.6. Finally, related work is discussed in §4.7 and we conclude in §4.8.

4.2 BACKGROUND

The background required for this chapter is mostly covered by the background of the previous chapter. We utilize initial algebra semantics as an implementation technique in this chapter, where our semantic functions are implemented as Σ -algebras for some signature (Σ). Throughout most of this chapter, we again use funcons as a semantic domain. To support exploratory programming, we utilize the approach of building sequential languages [31], and the approach to obtain exploratory programming for free introduced in Chapter 2. In this section we repeat the different components in more detail for completeness.

4.2.1 *Language Design, Implementation, and Evaluation*

There are different implementation strategies for DSLs [133]. One approach is to embed the implementation in a general-purpose language [85]. This general-purpose language then hosts the DSL. Such an embedded DSL (EDSL) has the benefit that no new parser is needed and tooling of the general-purpose language can be used for DSL programs. Sometimes it also supports using constructs from the host language in the DSL. An embedding can either be shallow or deep. With a shallow embedding, the operations of the DSL are directly mapped to operations in the general-purpose language. Thus, no abstract-syntax tree (AST) is built. This makes the implementation simpler but also more difficult to extend with analysis over programs written in the DSL. With a deep embedding, the DSL operations build terms to form an AST which can then be analyzed and evaluated.

An alternative implementation strategy for a DSL is to use a meta-language. Meta-languages often provide operations and tools to simplify DSL creation, such as reusing constructs or combining languages to build bigger languages. Erdweg et al. provide a framework for discussing and comparing meta-languages, tools and formalisms that support various forms of incremen-

tal language development [65]. In particular, the authors define the concepts of (modular) language extension, restriction, and unification, which they apply to both the syntax (concrete & abstract), static semantics, operational semantics and IDE services of languages. Extension occurs when a base language is extended by another language that has a dependency on the base language. Restriction is a special form of extension, where a language is restricted, making the new language a subset of the original language. Unification is the process of combining two independent languages with the help of glue code to unify the two languages. The paper also distinguishes between different forms of extension: no extension composition, incremental extension, and extension unification. In case a method does not support extension composition, it is impossible to combine multiple extensions. For incremental extension, extension can be performed in layers where one extension extends the base and another extension extends the extensions, etc. With extension unification, two extensions are unified and the unification is used as the extension on a base language. In this chapter we adopt their terminology and use their framework as part of our evaluation.

4.2.2 *Programming Language Semantics*

The initial algebra semantics of Goguen et al. [76], concisely described by Mosses in [142], provides the formal foundation and terminology to our work. Initial algebra semantics captures the essential elements of many existing semantic specification formalisms, such as denotational semantics and attribute grammars. A multi-sorted signature (Σ) lays out the operators of a language in terms of a set of sorts — a set of symbols functioning as an index set, such as $\{\mathbf{int}, \mathbf{bool}\}$. A Σ -algebra assigns carrier sets to these sorts. When taking term-constructors as the carriers, we obtain the abstract syntax of the language. The algebra formed this way is initial in the class of Σ -algebras. Due to its initiality, there is a unique homomorphism from the initial algebra to any algebra in the class of Σ -algebras. This unique homomorphism to a Σ -algebra is the catamorphism for the Σ -algebra [130]. Algebras give meaning to the operators of a signature by assigning a semantic function to each. By initiality, any abstract syntax can

be mapped to the semantics described by an algebra. Formulated differently, there is a systematic way to apply any algebra to the terms of a language.

The component-based approach to operational semantics presented in [144] is centered around reusable definitions of the *fundamental constructs* of (general-purpose) programming languages – referred to as *funcons* for short. An example funcon term is `print(integer-add(1,2))`, which outputs the result of $1 + 2$ and is constructed using the `print` and `integer-add` funcons. Throughout the text, funcons are indicated with a maroon color, except when used within code snippets. As explained in [29], ‘micro-interpreters’ can be generated from funcon definitions. The micro-interpreters are compositional evaluation functions expressing the behavior of an individual funcon that can be generated and compiled separately. In this chapter, we leverage the generality of the Funcons-Beta library [147] to be able to express the semantics of language constructs in a shared base language. Effectively, the generated micro-interpreters for funcons are applied as the constructs of an EDSL.

4.2.3 *Incremental and Exploratory Programming*

Exploratory programming [99, 174] is a style of programming in which the goal worked towards is open and by experimenting with code this goal evolves. This style of development constitutes the creation of different variants for experimentation and possibly discarding (some of) these variants during the process as well. It is thus a volatile style of development. Exploratory programming is in a limited form supported by incremental programming environments such as read-eval-print loops (REPLs) and notebooks. Incremental programming supports the submission of small snippets of code to obtain immediate feedback, resulting in a tight feedback loop which is useful during prototyping. However, these environments generally do not have first-class support for the exploration of multiple variants simultaneously nor for managing explorations that can be discarded. Previous work [31] provides a principled approach to (defining and developing) REPL interpreters. The approach involves adapting an existing language to a ‘sequential’ variant that naturally supports

incremental programming. Sequential languages are defined as languages in which any two programs can be sequenced together to form a new program, and the semantics of the sequence is identical to the composition of the semantics of the two programs in isolation.

For sequential languages, tooling for incremental programming such as REPLs, Jupyter Notebooks [107], and even exploratory programming environments, as demonstrated in Chapter 2, can be obtained for free. In this chapter, we apply the idea of sequential languages to support exploratory programming in our *meta-language* $iCoLa^+$ (i.e. exploratory language development) and to obtain REPL interpreters for $iCoLa^+$ and the defined *object languages* that support exploratory programming via the work presented in Chapter 2.

4.3 EXTENDED COMPOSITIONAL DEFINITIONS

In this section we generalize the formal model with support for an arbitrary amount of semantic domains and support for concrete syntax definitions for operators.

To generalize our approach to an arbitrary amount of semantic domains, we update the notion of an operator set to a family as follows:

Definition 4.3.1. Let D be a set of domains. Every domain gives rise to a set of valid terms in that domain, denoted with A_d for a given domain $d \in D$. An operator set O_D is a family $\langle O_d \rangle$ indexed by D . O_d is the set of operator symbols with a translation to the semantic domain d , i.e. there exists a function $A_d^{|o|} \rightarrow A_d$ for every operator symbol $o \in O_d$, where $|o|$ is the arity of the operator symbol o .

As a notational convenience, we use O to refer to all operator symbols, i.e. $O = \bigcup \{O_d \mid d \in D\}$.

When all operator symbols in an operator set have a translation to a specific domain, we call the set complete with respect to the domain.

Definition 4.3.2. An operator set, O_D , is complete with respect to a domain $d \in D$ iff all operator symbols have a translation to the semantic domain d .

We also update our language definition to handle the arbitrary domains and extend the language definition with a (concrete) syntax component. Syntax is defined at the language level instead of the operator level, because operator definitions are immutable. Syntax, however, can vary widely for the same operator between languages. As defined in the previous chapter, the \vec{o} operation refers to the operand locations of an operator.

Definition 4.3.3. A language is a structure $\langle T, S, G, I, P \rangle_{O_D}$ in terms of O_D , with

- $T \subseteq O$ being the set of top-level operator symbols;
- S is a family $\langle S_{o \in O, w \in \vec{o}} \rangle$ of sets indexed by $O \times \mathbb{N}$. $S_{o,w}$ is the set of operator symbols assigned to the operand position w of operand o ;
- G is a family $\langle G_{o \in O, w \in \vec{o}, d \in D} \rangle$ of functions indexed by $O \times \mathbb{N} \times D$. $G_{o,w,d}$ is the specialization function $O \times A_d \rightarrow A_d$ in the domain d for operators assigned to the operand position w of operand o ;
- I is a family $\langle I_{t \in T, d \in D} \rangle$ of functions indexed by $T \times D$. $I_{t,d} : A_d \rightarrow A_d$ denotes the top-level initialization function for the specific top-level operator in the given domain;
- P is a family $\langle P_{o \in O} \rangle$ of sets indexed by O . $P_o \subset (\Sigma \cup \vec{o})^*$ is the singleton set representing the syntax rule that produces the operator identified by the operator symbol o , and Σ is a set of terminal symbols. P_o thus represents a production rule for the operator o with symbols being terminals (Σ) and operator locations (\vec{o}).

The update to language composition (Definition 3.3.1) is trivial with a biased operator for syntax rules and the empty rule (ϵ) as the neutral element. As our bias, we pick a right-biased operator to give priority to syntax rules introduced by languages that occur on the right side of a composition.

In contrast to productions as used in the definition of a context-free grammar, our definition does not allow non-terminal to be explicitly defined by the user. Instead, for every operator and for all operand locations a non-terminal is generated. Inside a

production definition, only the operand locations and terminal symbols are available. It is thus impossible to refer to an operator directly in the definition of a production. Instead, this is done implicitly by referring to an operand location, which then holds for all the operators assigned to that operand location. To make this more concrete, we detail the algorithm to extract a context-free grammar from an *iCoLa*⁺ specification.

With the used syntax definition, a context-free grammar $G = \langle V, A, P, S \rangle$ can be generated using the algorithm specified in Algorithm 1. The process is as follows: We take the alphabet to be all operator symbols in the operator set, all operand locations, all terminal symbols, and add a symbol to represent the top-level. The set of terminals in the CFG correspond to the set of terminals used in the syntax rules. Productions for operand location non-terminals are obtained by creating an alternative for every non-terminal corresponding to the operators assigned to the location (in the algorithmic description, we use *choice* to combine alternatives into one production). Productions for operator symbols are obtained by taking the syntax rules and turning them into productions by replacing the location placeholders with their corresponding generated non-terminal. Finally, productions for the top-level non-terminal are obtained by creating an alternative for every non-terminal corresponding to the operators assigned to the top-level, and we denote the top-level as the distinguished element.

To illustrate this process we use the lambda language as a running example. The operators and constraints of our lambda language were defined as follows, with the top-level consisting of the operators assigned to the *Expr* sort.

$$\begin{array}{ll}
 & Var_{\mathcal{O}} \in Expr \\
 & App_{\mathcal{O}} \in Expr \\
 & Abs_{\mathcal{O}} \in Expr \\
 Var_{\mathcal{O}} : VarVar & String \subseteq VarVar \\
 Abs_{\mathcal{O}} : AbsVar \times AbsBody & String \subseteq AbsVar \\
 App_{\mathcal{O}} : AppAbs \times AppArg & Expr \subseteq AbsBody \\
 & Expr \subseteq AppAbs \\
 & Expr \subseteq AppArg
 \end{array}$$

Algorithm 1 Algorithm to turn meta-productions into a context-free grammar

```

1: function LANGToCFG( $\langle T, S, G, I, P_m \rangle_{O_d}$ )
2:    $A \leftarrow$  terminals  $P_m$ 
3:    $L \leftarrow \{o_w \mid o \in O, w \in \vec{\sigma}\}$   $\triangleright$  Create non-terminals for all
      operand locations
4:    $V \leftarrow O \cup L \cup \{TopLevel\} \cup A$ 
5:    $P \leftarrow \{\}$ 
6:    $S \leftarrow \{TopLevel\}$ 
7:   for  $o_w \leftarrow L$  do  $\triangleright$  Create productions for the operand
      location non-terminals
8:      $P \leftarrow P \cup \{(o_w, \text{reduce } G_{o,w} \text{ using choice})\}$ 
9:   end for
10:  for  $p \leftarrow P_m$  do
11:     $P \leftarrow P \cup \text{concrete } p$   $\triangleright$  Replace location placeholders
      with corresponding operand location non-terminal
12:  end for
13:   $P \leftarrow P \cup \{(TopLevel, \text{reduce } T \text{ using choice})\}$ 
14:  return  $\langle V, A, P, S \rangle$ 
15: end function

```

We extend this with the following production rules. The integers here refer to the operand locations of the operator. Thus 0 in P_{var} refers to the *VarVar* operand location.

$$\begin{aligned}
\langle P_{Var} \rangle &::= \langle 0 \rangle \\
\langle P_{Abs} \rangle &::= \langle 0 \rangle \rightarrow \langle 1 \rangle \\
\langle P_{App} \rangle &::= \langle 0 \rangle \langle 1 \rangle
\end{aligned}$$

With these rules we obtain the following values for the V and A sets.

$$\begin{aligned}
V &= \{Var, Abs, App, Var_0, Abs_0, Abs_1, App_0, App_1, TopLevel, -, >\} \\
A &= \{-, >\}
\end{aligned}$$

We then create productions from the syntax rules, with operand location productions first.

$$\begin{aligned}
\langle Var_0 \rangle & ::= \text{String} \\
\langle Abs_0 \rangle & ::= \text{String} \\
\langle Abs_1 \rangle & ::= \langle Var \rangle \mid \langle Abs \rangle \mid \langle App \rangle \\
\langle App_0 \rangle & ::= \langle Var \rangle \mid \langle Abs \rangle \mid \langle App \rangle \\
\langle App_1 \rangle & ::= \langle Var \rangle \mid \langle Abs \rangle \mid \langle App \rangle
\end{aligned}$$

We continue by filling the locations in the syntax rules turning the rules into productions for the CFG, and define the top-level production. The result of this process is a CFG for our definition of the lambda calculus.

$$\begin{aligned}
\langle Var \rangle & ::= \langle Var_0 \rangle \\
\langle Abs \rangle & ::= \langle Abs_0 \rangle \rightarrow \langle Abs_1 \rangle \\
\langle App \rangle & ::= \langle App_0 \rangle \langle App_1 \rangle \\
\langle TopLevel \rangle & ::= \langle App \rangle \mid \langle Abs \rangle \mid \langle Var \rangle
\end{aligned}$$

4.4 IMPLEMENTATION

In this section we discuss our DSL implementation of *iCoLa*⁺. The DSL is tool-oriented in the sense that *iCoLa*⁺ is by itself not executable, it needs to be embedded within a larger tool that orchestrates different components, one of which is *iCoLa*⁺. The tool-oriented design enables the construction of a pyramid abstraction, displayed in Figure 4.4, in which the different layers are operated by different kind of users. Users who operate in the lower parts of the pyramid require more expertise and provide abstractions as foundations for the users at higher-levels.

We start this section by explaining the design of the meta-language, then we detail the internal implementation, and we finalize by demonstrating the embedding of *iCoLa*⁺ inside tooling.

4.4.1 The *iCoLa*⁺ Language

4.4.1.1 Operators

Operators are defined using the `operator` keyword and require a unique name and a variable amount of operand locations. Operand locations are identified by names and these names must be unique among the locations for an operator, but not among locations across operators.¹ The following example demonstrates the definition of the three operators present in the lambda calculus.

```
operator Var : Var
operator Abs : Var Body
operator App : Fun Arg
```

The name before the `:` symbol indicates the name of the operator. The names after the `:` symbol are the names for operand locations. In case of our definition of the `Abs` operator, there are two locations: `Var` and `Body`.

Certain operators have operand locations that can take a variable number of values when instantiating the operator. An example of such an operator is a list. To support this, operand locations can have a modifier indicating the degree of values an operand location accepts. There are three modifiers currently available: the `+` modifier, indicating one or more values; the `*` modifier, indicating zero or more values; and the `?` modifier, indicating zero values or one value. For example, a list operator can be defined as follows in *iCoLa*⁺.

```
operator List : Item*
```

Indicating that the `Item` location can have zero or more values.

4.4.1.2 Operator Semantics

Operator semantics is given by translating an operator to a chosen semantic domain. Every semantic domain is uniquely identified by its name. The name of a semantic domain identifies a translation function in the DSL. We discuss how domain definition are

¹ Essentially, operators introduce a namespace for the locations. Since operator names are unique, we retain the uniqueness of operand locations as required by the conceptual approach.

defined in detail in §4.5.8. In our example, we again utilize the lambda calculus with a translation to the funcons domain.

```
funcons Var = bound(@Var)
funcons Abs = function closure scope(bind(@Var, given), @Body)
funcons App = apply(@Fun, @Arg)
```

The example starts with the name of the semantic domain — funcons — to indicate that the translation function is into the semantic domain of funcons. After the name of the semantic domain, the operator being translated is identified by its name, and is followed by the = operator to indicate the start of the body of the translation function. The syntax available in the body of a translation depends on the domain in which the translation function is being defined. A domain definition is free in its choice of syntax as long as it supports naming holes, which is achieved with the @ operator in the funcons domain. The names for holes correspond to operand locations for the operator being translated. The funcon semantic domain uses funcon terms as the concrete syntax as used by the PlanComps project. Essentially, a domain definition can define its own DSL inside the *iCoLa*⁺ DSL.

4.4.1.3 Language Definitions

Languages are defined via the language keyword and can contain sort constraints, parser definitions, and operator specialization code. To illustrate, we construct a variant of the lambda calculus in steps, starting with the constraints defining the language.

```
sort Expr
language Lambda
{ {Var, Abs, App} < Expr
, { String } < Var[Var]
, { String } < Abs[Var]
, Expr < Abs[Body]
, Expr < App[Fun, Arg]
, Expr < TopLevel
}
```

We first define the Expr auxiliary sort. Then a new language, named Lambda, is introduced. Inside the language definition, we start with the {Var, Abs, App} < Expr constraint. This states that the Expr sort must contain those three operators to be a correct instantiation of the Lambda definition. We continue by

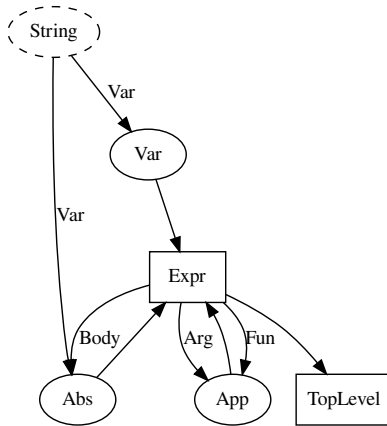


Figure 4.1: Constraint graph for the lambda calculus example. Ellipses denote operators, dashed ellipses denote built-in operators, and rectangles denote auxiliary sorts. Edges denote assignment of operator to the operand location determined by the edge label.

assigning the `String` operator to the operand location `Var` for the `Var` and `Abs` operators. The syntax operator `[name]` is used to reference the operand location identified by name for the specified operator. This is followed by distributing the `Expr` sort over the operand locations where expression can occur, and assigning the `Expr` sort to the top-level. With this definition, we get the constraint graph displayed in Figure 4.1 for our `Lambda` definition.

In language definitions there is no separate syntax between operator assignments and sub-sort constraints. Instead, operator assignments are defined as sub-sort constraints using an anonymous sort — identified by the inline set notation.

The current definition is not concise and duplicates certain structures. To improve this, *iCoLa⁺* has several forms of syntactic sugar. The first form is focused on usage of a specific sort in multiple constraints can be merged via sequencing. In our example, this occurs with the `String` operator and the `Expr` sort. We can update our definition as follows.

```

sort Expr
language Lambda
{ {Var, Abs, App} < Expr
, { String } < Var[Var] ; Abs[Var]
, Expr < Abs[Body] ; App[Fun, Arg] ; TopLevel
}

```

The sequence operator (;) groups sorts together and applies the constraints as if it were individual constraints. In the updated definition, sequencing occurs on the right-hand side of the < operator. Sequencing can also occur on the left-hand side. Another form is focus on constraints that use all operand locations of an operator. In our example this occurs within the constraints referencing Var[Var] and App[Fun, Arg]. Instead of naming all locations, an empty indexing expression [] can be used. With this form of syntactic sugar, our example definition can be updated as follows.

```

sort Expr
language Lambda
{ {Var, Abs, App} < Expr
, { String } < Var[] ; Abs[Var]
, Expr < Abs[Body] ; App[] ; TopLevel
}

```

Having reduced our definition, the Expr subsort is only used in one place, and seems redundant. We could thus update the definition by removing this sort.

```

language Lambda
{ {Var, Abs, App} < Abs[Body] ; App[] ; TopLevel
, { String } < Var[] ; Abs[Var]
}

```

However, removing such an auxiliary sort makes it more work to add a new operator that is usable at the same operand locations as the three existing operators. Whether to use an auxiliary sort depends on the intended usage of the language and the used operators. There is no correct way and one choice does not restrict future usage of a defined language, since these structural choices can be reverted via refinements and by introducing new constraints.

Having finalized our constraints for the lambda calculus, we move on to the concrete syntax. Concrete syntax is added to a language by defining syntax rules using the ::= symbol. Our

lambda calculus language can thus be extended with concrete syntax as follows.

```
Lambda
{ Var ::= Var
, Abs ::= '\\\ ' Var "->" Body
, App ::= Fun Arg
}
```

Instead of using the language keyword to introduce a new language, we extend the existing definition by referring to the name identifying the language. In the extension, we define syntax rules for the three operators in the lambda calculus. We could have also defined the syntax rules for the three operators in three separate extensions. The left-hand side of the `::=` symbol identifies the operator for which the syntax rule is being defined, and the right-hand side contains the actual body of the syntax rule. The right-hand side has access to the locations of the current operator by referring to the name identifying the location. Besides locations, syntax rules can contain literal strings — enclosed between `" "` — and literal characters — enclosed between `' '`. When an operand location takes a variable number of values, parsing modifications can control the parsing behavior. For example, a parser for our previous list operator can be defined as follows: `list ::= Item{,}`. In this example, the `{,}` syntax denotes that the items are parsed separated by a separation character, which is in this case a comma. In case no parser modification is given, it will parse multiple items without any separation character separating them.

Our current definition of the lambda calculus can be slightly extended by showing evaluation results via the addition of initialization semantics to the language. In the DSL, specializing the `TopLevel` sort results in such initialization code.

```
Lambda
{ funcons TopLevel when Expr => print @this
}
```

This example specializes the `TopLevel` sort for the `funcons` domain when the operator currently bound to the top-level is part of the `Expr` sort. The specialization wraps the current term in a print statement, printing the result of the evaluation. The current term being specialized in the semantic domain is referenced

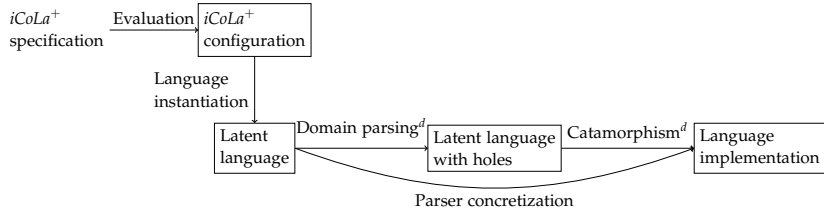


Figure 4.2: Overview of the $iCoLa^+$ evaluation pipeline. Domain parsing and the catamorphism are parameterized by a domain definition.

using this. Without a when expression, specialization is always applied to all operators assigned to the specialized location. In our example, we might extend the language with statements and assign statements to the top-level, but not to the Expr sort. With the current specialization, such operators would not be wrapped in a print statement. By removing the when expression, such statements would be wrapped in a print statement. In contrast to the formal model, in the DSL we make no distinction between initialization and specialization code. Instead, initialization is achieved by specializing the `TopLevel` sort, which is built-in.

4.4.2 Internal Representation

$iCoLa^+$ is implemented in Haskell and in this section we detail the internal representation and the evaluation pipeline that turns an $iCoLa^+$ specification into an executable language given a language definition and a semantic domain. Figure 4.2 gives an overview of this process.

4.4.2.1 Operators

Operators are implemented as functors — type transformations that come with a function (`fmap` in Haskell) to lift functions from the original type to functions on the transformed type. Using functors, we can use the same data type for the different representations of an operator. When an operator is defined in the DSL, it has a string representation. When a language is instantiated, operators have a fix-point representation, allowing us to represent terms of the defined language.

```

data Operator a = Operator ID [a]
    | OBuiltIn (BuiltInOp a)

data BuiltInOp a = OTuple [a]
    | OInt Int
    | O0String String
    | OBool Bool
    {- ... -}

```

The `Operator` constructor is used for operator definitions. The remaining constructor is used for the built-in operator data type, of which a selection of the constructors is shown.

4.4.2.2 *Semantic Domains*

A semantic domain is represented by a name, a parse function, an algebra definition, and an instance of the substitution type-class.

```

data Domain a =
    Domain String (String → Either String a) (BuiltInOp a → a)

class Substitution a where
    subst :: M.Map ID a → a → a

```

The parse function takes the body of a semantic function and translates it into an internal representation with holes, or returns an error with a descriptive message. When successful, the algebra is applied on this internal representation, which translates built-in operators to the semantic domain. The substitution type-class implements the replacement of named holes with the terms to which the names are mapped, and is used to automatically translate the `Operator` constructor to the semantic domain. With this setup, an actual domain definition is simple.

```

funconsDomain :: Domain Funcons
funconsDomain = Domain "funcons" Funcons.parse funconsAlg

funconsAlg :: BuiltInOp FT.Funcons → FT.Funcons
funconsAlg (OInt i) = FT.int_ i
funconsAlg (O0String s) = FT.string_ s
funconsAlg (OBool b) = FT.bool_ b
funconsAlg (OTuple t) = FT.tuple_ t

instance Substitution Funcons where
    subst = applyFuncon

```

The `Funcons.parse` function parses the concrete syntax of funcons into a `funcons` term with holes, and is provided by the `funcons tools` package.² The `applyFuncon` function substitutes named holes with the term to which the name is mapped, and is provided by `us`. As previously mentioned, the definition of an algebra is simple, primarily because the complexity of a translation is internal to $iCoLa^+$, and shared between all domains.

4.4.2.3 Parser Concretization

Within $iCoLa^+$, language composition is a fundamental operation on languages. Having extended languages with concrete syntax, the concrete syntax specifications are also combined. It is well known that combining two unambiguous context-free grammars does not necessarily result in an unambiguous grammar. Therefore, to handle ambiguity in both syntax rules and the generated context-free grammars, we utilize Generalized LL (GLL) parsing [193]. Syntax rules are implemented by translating them to combinators defined by the GLL combinators library [30, 208], and follows the process as outlined in Algorithm 1.

Essentially, there is a mapping for every construct available in the syntax rules to a combinator in the GLL library. For example, parsers for operand locations are achieved by mapping to the `<|>` combinator, denoting choice; and operand locations with a modifier, such as `+`, can parse multiple values, and are mapped to the `multiple` or `multiple1` combinator, depending on the modifier being `*` or `+`, respectively.

To handle the parse results of operand location modifiers, the `OTuple` constructor is used. When the `?` modifier is used, the result is either a tuple with one element containing the result or a tuple with zero elements. In case of the `+` and `*` modifiers, the `OTuple` contains all the parsed values. Semantic domains thus must support operations on tuples. For example, the translation to the funcon domain for the `List` operator is as follows.

```
funcons List = list tuple-elements @Item
```

The `tuple-elements` extracts a tuple into a sequence of values, which is accepted by the `funcons list` constructor. The method in which a semantic domain supports operation on tuples is not

² <https://hackage.haskell.org/package/funcons-tools>

fixed. In case of the funcons domain, this needs to be explicitly encoded in the translation functions. Alternatively, a semantic domain can encode this internally without requiring the unpacking of tuples to happen inside translation functions. Then, for example, the `list` constructor would accept a tuple as an argument, and would simply be the following.

```
funcons List = list @Item
```

The choice of encoding has both advantages and disadvantages. In case of funcons, we sometimes observe many calls to `tuple-elements`. In funcons, sequences are not actually values. As a result, tuples need to be unpacked into sequences operationally. Alternatively, when sequences are directly supported and a sequence needs to be seen as one value, one needs to wrap it into a tuple. A possible mitigating solution is by improving the syntax of the domain. For example, using a Python-like unpacking operation, such as `*@Item` would already be a significant improvement to the clutter introduced by the many calls to `tuple-elements`.

4.4.3 *Environment Definitions*

Programmers interact differently with different programming languages. To accommodate this, *iCoLa*⁺ is designed with user-definable environment definitions in mind. Figure 4.3 gives an abstract view of environment definitions, which consists out of two environments: the meta-environment and the language environment. A language designer interacts with a meta-environment and a user of the defined language interacts with the language environment. These users can be different. For example, a language engineering expert and a domain expert for which the language is designed. A user can also function as both types simultaneously. The method of interaction is determined by environment definitions, and can vary depending on the type of user. An environment definition orchestrates the interaction between the different users with their respective environments, and in case of the meta-environment determines the capabilities via the selection of domain definitions.

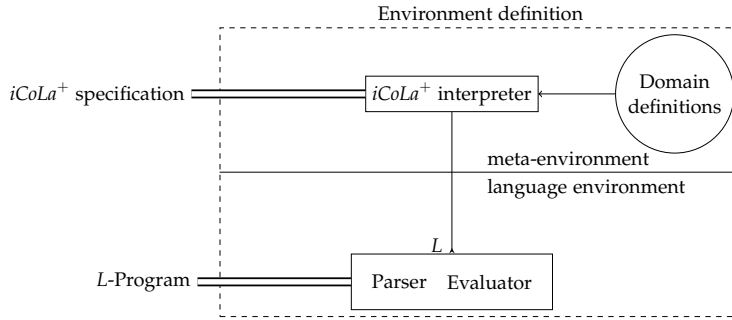


Figure 4.3: Visual view showing the embedding of $iCoLa^+$ inside an environment definition. Normal arrows denote a dependency. Stealthed arrows denote generation. Pipes denote abstract interaction between two components. The concrete interactions patterns are determined by the environment definitions and therefore unknown to $iCoLa^+$. Inspired by [104].

Inside an environment definition, there is a dependency of the language environment on the meta-environment in the form of generated language implementations. Internally, the `instantiateLanguage` function is used to obtain a language implementation for a given language and a given domain.

```
instantiateLanguage :: String → Domain a
                  → ICoLa (String → FOperator, FOperator → a)
```

The first parameter to the function is the name of the language being instantiated, and the second parameter is the domain for which the language is being instantiated. The result of this function is a parser and an evaluator, which can be used by the environment definition to connect to user interaction in whatever way fit. The parser and evaluator are separate instead of composed to give flexibility to the environment definition.

When instantiating a language, we require that the language is complete with respect to the chosen domain (Definition 4.3.2). If not, an error is thrown.

Not every domain definition in an $iCoLa^+$ specification will be used by an environment definition. Therefore, $iCoLa^+$ only checks the correctness of the domain when a language is instantiated for that domain. Therefore, specification do not need to change when used in a defined environment that does not support a domain contained in the specification.

4.4.4 The Interaction Layers of $iCoLa^+$

The $iCoLa^+$ implementation can be divided in three layers: the DSL for language and operator definitions, the environment definitions, and the domain definitions. The aim of this separation is to allow different kind of users with different expertise and knowledge to interact with $iCoLa^+$. The layers form the aforementioned pyramid abstraction as displayed in Figure 4.4.

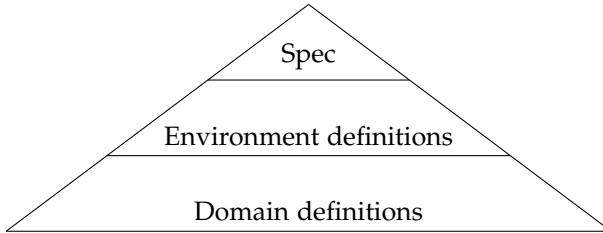


Figure 4.4: Abstraction pyramid showing the different interaction layers of $iCoLa^+$.

Most users will interact with the top layer of the pyramid, since languages and operators are defined at that layer. Users wanting to integrate language environments into tooling or wanting to work on user interfaces for language development will interact with the middle layer. Language engineering experts that want to experiment with alternative approaches to semantic specifications will interact with the bottom layer. Of course, a user can interact with multiple layers as well, but the intention behind the abstraction pyramid is that it is, in the general case, not needed. The different layers thus also require different levels of understanding of the system. Interaction with the *Spec* layer requires no Haskell knowledge and no knowledge of the implementation; it does require understanding of the semantic domains being offered to the user, and the conceptual idea of $iCoLa^+$ operator and sort definitions. Users that interact with $iCoLa^+$ via the *Environment definitions* layer need to understand the basics of Haskell. They do not need to understand the actual details of the evaluation pipeline. Users interacting with the *Domain definitions* layer need to have an intermediate understanding of Haskell, and need to understand the basics of the $iCoLa^+$ evaluation pipeline as shown in Figure 4.2, e.g., relating to the usage of catamorphisms.

4.5 A DEMONSTRATION OF *iCoLa*⁺

In this section we evaluate *iCoLa*⁺ by conducting an extended exact replication of the evaluation performed on *iCoLa*. We opted for this kind of evaluation to enable us to focus on the extensions proposed in this chapter, and to be able to compare the implementation introduced in this chapter to the already existing implementation introduced in the previous chapter, and by extending *iCoLa*⁺ via the provided extension points.

As part of the evaluation, we construct the following three languages via the composition, extension, and refinement of existing languages and language fragments — language definitions that are not executable by themselves. *Imp* [181], a simple imperative language; *SIMPLE* [182], a more complex procedural language; and *MiniJava* [8], a strict subset of the Java language. These languages are chosen because they have their semantics described in terms of funcons as part of the case studies for the PlanCompS project.³ We have various reasons for picking languages that already have their structure and semantics expressed. With this choice, we are able to demonstrate that our approach is applicable to already existing language definitions, and is effective as shown by taking existing language definitions and turning them into the compositions of smaller languages. In addition, we are able to show that our approach promotes reusability by reusing language definitions within new definitions. It is incremental because the chosen languages are defined in an incremental and step-wise manner, and is flexible by showing that language design choices do not restrict future usage of language definitions. Besides the replication of the previous evaluation, we also look at defining DSLs with less obvious operational semantics in *iCoLa*⁺ by defining a DOT-like language.

The structure of this section is based on the abstraction pyramid introduced in the previous section. §§4.5.1-4.5.5 correspond to the *Spec* layer; §4.5.7 corresponds to *Environment definitions* layer; and §4.5.8 corresponds to the *Domain definitions* layer.

³ <https://plancomps.github.io/CBS-beta/docs/Languages-beta/index.html>

```

Language ImpArith
{ {Int, Id, Div, Add, AParen} <
  Aexpr
, {String} < Id[]
, Aexpr < Add[] ; Div[] ;
  AParen[]
, funcons Div => checked @this
}

Language ImpBExpr
{ { Bool, Leq, Not, And, BParen
  } < Bexpr
, Bexpr < Not[] ; And[] ;
  BParen[]
, Aexpr < Leq[]
}

Language ImpStmt
{ {Assign, If, While, Block} <
  Stmt
, { String } < Assign[Id]
, Aexpr < Assign[Expr]
, Bexpr < While[Cond] ; If[Cond
  ]
, Stmt < Block[]
, { Block } < If[True, False] ;
  While[Body]
}

Language ImpProgram
{ { String } < SIdList[Id]
, { SIdList } < IdList[] ;
  Program[Ids]
, { IdList } < IdList[Rem] ;
  Program[Ids]
, Stmt < Program[Program]
, { Program } < TopLevel
}

```

Figure 4.5: Structure definitions for the language fragments used in the definition of the *Imp* language.

4.5.1 Specification of *Imp*

We define *Imp* as the composition of the following four languages:

```
Language Imp = ImpArith <> ImpBExpr <> ImpStmt <> ImpProgram
```

The composition is comprised of a simple arithmetic language with support for integer addition and division; a boolean expression language with support for less-than-equal comparison and (binary) conjunction; a statement-language containing if-statements, while-statements, and sequencing of statements; and a program language that unifies these languages together by defining the top-level in accordance to the top-level of *Imp* (in that order). The structural definition of these languages are displayed in Figure 4.5, and the concrete syntax definitions for operators used in the fragments are displayed in Figure 4.6.

Except for the definition of *ImpProgram*, the top-level is not constrained. These language definitions are therefore not executable by themselves due to there being no entry points. Such languages will be referred to as language fragments. The *ImpArith* fragment

```

ImpArith
{ Add ::= @Left '+' @Right
, Div ::= @Left '/' @Right
, AParen ::= '(' @Expr ')'
, Id ::= @Var
}

ImpBExpr
{ Not ::= '!' @Expr
, Leq ::= @Left "<=" @Right
, And ::= @Left "&&" @Right
, BParen ::= '(' @Expr ')'
}

ImpStmt
{ While ::= "while" '(' @Cond
          ')' @Body
, If ::= "if" '(' @Cond ')'
        @True
        "else" @False
, Block ::= '{' @Stmt @Rem '}'
, Assign ::= @Id '=' @Expr ';'
}

ImpProgram
{ Program ::= "int" @Ids ';'
          @Program
, IdList ::= @Id ',' @Rem
, SIdList ::= @Id
}

```

Figure 4.6: Concrete syntax extensions to the language fragments used in the definition of the *Imp* language.

defines specialization code over the *Div* operator, wrapping the divide in a check. When division by zero occurs, the program is terminated following the application of the funcon **checked** as in Chapter 3. This behavior is not directly encoded in the semantics of the *Div* operator, because other languages handle this differently, for example by throwing an exception. Another observable is the usage of the *Aexpr* auxiliary sort by the *ImpBExpr* language in its definition ($Aexpr < Leq[]$) without assigning operators to this sort. This makes *ImpBExpr* dependent on a language that does assign operators to the *Aexpr* sort, such as *ImpArith*.

ImpBExpr can also be defined independently of *ImpArith* by not using the the *Aexpr* sort, but use a glue language to glue the two fragments together. We define a *weaker* variant of the *ImpBExpr* language that is independent of *ImpArith* as follows.

```

language ImpBExpr-
{ {Bool, Leq, Not, And } < Bexpr
, Bexpr < Not[] ; And[]
}

```

And we recover our original definition by unifying this language with *ImpArith* via a glue language.

```

language ImpBGlue = { Aexpr < Leq[] }
language ImpBExpr = impArith <> impBExpr- <> ImpBGlue

```

4.5.2 Specification of *SIMPLE*

In the previous section we saw how *Imp* was defined via the composition of smaller languages. This component style of development is one the basis to simplify language development in *iCoLa*⁺. Another basis is reusing existing language definitions to define new languages. To demonstrate this, we utilize *Imp* and the language fragments used to define it in the definition *SIMPLE* and *MiniJava*. Even when the definition of *Imp* does not directly correspond to the definition of the to be defined language.

There are two adaptations required to *Imp* to define *SIMPLE* as an extension of *Imp*: removing the top-level definition of *Imp* and removing the distinction *Imp* makes between arithmetic and boolean expressions. Note that *Imp* variables can only occur inside arithmetic expressions. For the top-level adaptation, we opt to define the base using the language fragments of *Imp* without the *ImpProgram* fragment. For the second adaptation we define a new language that glues the two different expressions into a new sort:

```

sort Expr
language UnifiedExpr
{ Aexpr ; Bexpr < Expr
, Expr < Leq[] ; Add[] ; Div[] ; While[Cond] ; If[Cond]
}

```

We first constrain the new *Expr* sort with both the *Aexpr* and *Bexpr* sort. Then we constrain the operand locations of *Imp* operators which also occur in *SIMPLE* and use expressions with the new *Expr* sort. This language definition removes structural choices of *Imp* to align with *SIMPLE*. With the unification language, we can reuse the fragments to define a base for *SIMPLE*:

```

language SimpleBase = UnifiedExpr <> ImpArith <> ImpBExpr <>
    ImpStmt

```

By utilizing existing language components we have already obtained a part of the *SIMPLE* language. We can now take this base and extend it with the constructs that are present in *SIMPLE* but not in *Imp*. A subset of these constructs is displayed in Table 4.1. The table is not exhaustive. Most operators that only occur within *SIMPLE* have been omitted for brevity. Operators are grouped to indicate their relation within a possible language fragment.

Table 4.1: The rows indicate operators used during the evaluation and the columns the constructed languages from the collection. The ● indicates that the operator is used as is; ◐ indicates that an operator is used with glue code; and ○ indicates that an operator is not used. A * next to an operator indicates that the concrete syntax for the operator is not identical between the languages.

	<i>Imp</i>	<i>MiniJava</i>	<i>SIMPLE</i>
Arith			
Addition	●	●	●
Division	◐	◐	◐
Substraction	○	●	●
Multiplication	○	●	●
Bool			
Negation	●	●	●
≤	●	○	●
And	●	●	●
Or	○	○	●
<	○	●	●
Statements			
If*	●	●	●
While	●	●	●
Assignments*	●	●	●
Input/Output			
Ouput*	○	◐	●
Input	○	○	●
Classes	○	●	○
Arrays			
Length*	○	●	●
Indexing	○	◐	◐
Exceptions			
Throw	○	○	●
TryCatch	○	○	●

4.5.3 Specification of *MiniJava*

To define *MiniJava*, we can reuse the definition of the base for *SIMPLE*. However, *MiniJava* requires one more step, because the less-than-equal operator does not occur in *MiniJava*. Therefore, we define a refinement which removes the less-than-equal operator from the *SIMPLE* base.

```
Language MiniJavaBase = refine SimpleBase { Leq }
```

Refinement in $iCoLa^+$ is achieved by defining a set of sort-constraints or operators that the refined language should not satisfy. So, in this example, the *MiniJavaBase* language cannot contain the assignment of *Leq* operator to any operand location. More refined refinements are also possible, for example refining over a sub-sort $Leq \subseteq Expr$, which states that the refined language should not have the *Leq* operator assigned to the *Expr* sort.

Building on our *MiniJava* base, we can add operators not present in *Imp*, such as classes and arrays. As was the case in Chapter 3, many operators used within *MiniJava* are also present within *SIMPLE*, but the usage is not always identical. Glue code helps to alleviate such differences.

Our earlier remark regarding variability applications in the context of teaching remain. With $iCoLa^+$, variability can now also be applied to the concrete syntax of the language. In this case, two languages might have the same semantics but different syntax that places emphasis on different aspects of the language or on supporting different mental models.

4.5.4 Object language variability

Language variability is not only useful for teaching purposes; it is also useful when designing a programming language. With $iCoLa^+$, different variants of a language can be defined and tested with relative ease. Multiple variants can exist side-by-side, making it easy to compare and contrast variations and gather feedback early, on both the concrete syntax and semantics, to include in the design process. Table 4.2 demonstrates some of the variability one can obtain with a relative small set of operators. The language definitions were defined in isolation or via composition.

Table 4.2: Table demonstrating a view from a session in the *iCoLa*⁺-shell, constructing several languages with a fixed-set of operators. Columns indicate the operators used during the evaluation and the rows are the languages constructed with (some) operators from the collection. The ● indicates that the operator is used as is; ◐ indicates that an operator is used with glue code; and ○ indicates that an operator is not used.

	Var	Abs	App _{cbv}	Addition	Int	Return	Call/cc	If	Throw	Catch
lambda	●	●	●	○	○	○	○	○	○	○
arithmetic	○	○	○	●	●	○	○	○	○	○
exceptions	○	○	○	○	○	○	○	○	●	●
proc	●	◐	●	○	○	●	○	○	○	○
lambda _{cbn}	◐	●	◐	○	○	○	○	○	○	○
functional	●	●	●	●	●	○	●	●	●	○
procedural	●	◐	●	●	●	●	○	●	●	●
procedural + functional	●	◐	●	●	●	●	●	●	●	●

For instance, *lambda_{cbn}* is defined by composing the *lambda* language with a language consisting (only) of glue-code that inserts the semantics of call-by-name using thunks [23].

```

Language cbnGlue
{ funcons App[Arg] ⇒ thunk @this
, funcons Var ⇒ force @this
}
lambdaCBN = lambda <> cbnGlue

```

In this definition, we again assume that all variables are assigned to thunked values. This is not always the case, e.g. in a procedural language with global variables. Type information can be used to distinguish variables based on whether their values are thunked. This, however, is not possible in our glue code definitions because glue code is context-free. Nonetheless, it can be realized within the semantic domain of funcons, as funcon terms are dynamically typed. The table shows an overlap between different languages and the two forms of variability in our approach: we can add new operators to existing languages and add new languages using existing operators, without modification of existing code.

4.5.5 *Exploratory Language Development*

The variability obtained in the previous section was achieved purely via incremental programming by introducing unique names for variants, as illustrated with the different lambda variants. In an exploratory setting, such variants can instead be explicitly defined as variants with the same name, supporting the utilization of both variants throughout an exploration. For instance, we could have defined our two lambda versions as two explicit variants by introducing a branch for both variants. An advantage of this approach is that we can experiment on both branches without having to duplicate our steps by replicating the actions executed on one branch automatically on the other branch, a concept we call mirroring. With mirroring, we can explore multiple branches simultaneously while operating on a single branch. A visual presentation of the idea of mirroring is displayed in Figure 4.7.

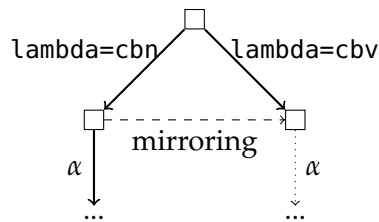


Figure 4.7: Visual idea of mirroring during exploratory programming, where a branch mirrors another branch explore different paths without duplication. α represents the execution of an arbitrary *iCoLa*⁺ program, boxes denote configurations, solid lines denote actions taken by the user, dashed lines denote meta-actions, and dotted lines denote actions automatically done by system.

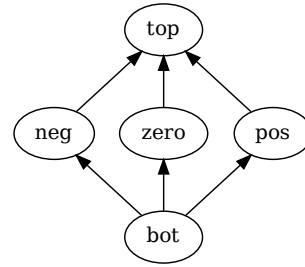
Mirroring is not the only advantage of first-class support for exploratory programming. With first-class support it is also trivial to experiment with different combinations of languages within different context, for example functional vs object-oriented, and switch between the contexts easily while also being able to compare the explorations. Furthermore, handling dead-ends during exploration is also supported by being able to go back to earlier points of the exploration. Since exploratory programming is an

```

digraph {
  top;
  bot;

  bot -> neg;
  neg -> top;
  bot -> zero;
  zero -> top;
  bot -> pos;
  pos -> top;
}

```



Listing 4.1: DOT program describing the lattice of signs of integers.

Figure 4.8: The visualization of the lattice of signs as described by the DOT program in Listing 4.1.

open ended task, such dead-ends are not unusual and having to restart a session, thus losing all context, hampers the experimentation. The support for exploratory programming within *iCoLa*⁺ is essentially achieved for free via the design of our DSL as a sequential language and the generic back-end introduced in Chapter 2.

4.5.6 DSLs in *iCoLa*⁺

So far, the defined languages have been variants of general-purpose languages. We now look at DSLs by implementing a DOT-like language. The DOT language is a DSL for describing the visualization of graphs, and is part of the Graphviz suite [64].

An example DOT program describing the lattice of integer signs is shown in Listing 4.1, which gives the visualization displayed in Figure 4.8. The DOT language has no computational elements. In this example we will extend a subset of the DOT language with for-loops. We have chosen the DOT language for two reasons. i): It is a small example that highlights certain pain points in *iCoLa*⁺ well, which we discuss in §4.6. ii): The operational semantics of the DOT language are not immediately clear, making it a good demonstration on how to capture such languages in *iCoLa*⁺.

The basic structure of the language is captured using the following operator definitions.

```
operator Graph : Name Component*
funcons Graph = scope({ "nodes" |-> allocate-variable(values),
                        "edges" |-> allocate-variable(values) },
                    seq(assign(bound-value "nodes", map-empty),
                        assign(bound-value "edges", set-empty),
                        effect tuple-elements(@Component),
                        tuple(@Name, assigned bound-values "edges",
                            assigned bound-values "nodes")))

operator Node : Name Annot?
funcons Node = assign(bound-value "nodes", map-override({ @n |->
    map-override(tuple-elements(@a), map-empty)}, assigned bound-
    value "nodes"))

operator Edge : Source Target
funcons Edge = assign(bound-value "edges", set-unite({(@s, @t)},
    assigned bound-value "edges"))

operator Annot : Annot
funcons Annot = @Annot

operator KV : Key Value
funcons KV = { @Key |-> @Value }

operator Program : Graph*
funcons Program = @Graph
```

A graph has a name and a sequence of components, which are comprised out of nodes and edges. The semantics of a graph is a tuple containing the name of the graph, the edges in the graph and the nodes in the graph. Nodes are modeled as maps mapping from the name of the node to its attributes, which we call annotations. Annotations are key-value pairs. Edges are modeled as sets of tuples, where a tuple denotes an undirected edge between the two elements.

With these operators, we define the first version of our DOT language.

```
language DOT
{ { String } < Node[Name] ; Graph[Name] ; Edge[] ; KV[]
, { KV } < Annot[Annot]
, { Node , Edge } < Graph[Component]
, { Annot } < Node[Annot]
, { Program } < TopLevel }
```

The current version of our language only captures undirected graphs. The DOT language also supports directed graphs. To support directed graphs, we introduce a new operator.

```
operator DGraph : Name Component*
funcons DGraph = scope({ "nodes" |-> allocate-variable(values),
                        "edges" |-> allocate-variable(values) },
                      seq(assign(bound-value "nodes", map-empty),
                          assign(bound-value "edges", set-empty),
                          effect tuple-elements(@Component),
                          tuple(@Name, assigned bound-values "edges",
                              assigned bound-values "nodes")))
```

The operator has the same semantics as the Graph operator. To differentiate between the two, we assign unconditional glue code to the two graphs.

```
DOT
{ { DGraph } < Program[]
, {Node, Edge} < DGraph[Component]
, {String} < DGraph[Name]
, funcons DGraph => variant("directed", @this)
, funcons Graph => variant("undirected", @this)
}
```

The glue code wraps the graphs in a variant with a tag indicating the type of graph. The type of graph affects the way the set of edges of a graph is interpreted. This change gives us a language in which both directed and undirected graphs can be specified. The semantics of our DOT language is the computation of lists of tagged-graphs. These tagged graphs can be taken by a layout algorithm to be placed on some canvas, or can be translated according to the Graphviz format.

We can extend our DOT language with computational components to simplify the definition of certain graphs. To do that, we extend the language with for-loops. The definition of the for-loop operator is given in Listing 4.2. A for-loop has two operands, one denoting the iteration and the other the body. The semantics of an iteration is given as a triplet, where the first element denotes the prelude and the stop condition, the second element denotes the pre-step, and the third element denotes the post-step. The semantics of the for-loop is then given by weaving these into a while loop. Using the for-loop extension, we can define the lattice of integer signs by the program in Listing 4.3, with the resulting diagram shown in Figure 4.8.

Listing 4.3: An example program in our variant of the DOT language encoding the lattice of integer signs (see Figure 4.8).

```
digraph "lattice" {
  node "bot"
  node "top"

  for "x" in ["neg", "zero", "pos"] {
    node $"x"
    $"x" -> "top"
    "bot" -> $"x"
  }
}
```

Listing 4.2: Definition of the For and In operators as used in our proposed extension to the DOT language.

```
operator In : Var Seq*
toFuncons(In v s) = tuple(tuple(bind("__i", scope(bind("__a", alloc
  values), seq(assign(bound "__a", 1), bound "__a"))),
  abstraction(is-less-or-equal(assigned bound "__i", length(tuple
  -elements @s))), abstraction bind(@v, index(assigned(bound-
  value("__i")), tuple-elements @s)), abstraction assign(bound("
  __i"), integer-add(1, assigned bound "__i"))))

operator For : Iter Body*
toFuncons(For it b) = scope(first tuple-elements first tuple-
  elements @it, while (enact second tuple-elements first tuple-
  elements @it, seq(scope(enact second tuple-elements @it, effect
  tuple-elements @b), enact third tuple-elements @it)))
```

4.5.7 Environment Definitions

The session as described by Table 4.2 was performed in the *iCoLa*⁺-shell. The *iCoLa*⁺-shell is a construction of the original *iCoLa*-shell as an environment definition with support for the aforementioned exploratory programming. In the *iCoLa*⁺-shell, users can define operators and languages, and commit a language which results in a REPL for the defined language. After experimenting with this language, they can return back to their session in the *iCoLa*⁺-shell, adapt the language definition and then commit the newly defined language. Furthermore, within

the *iCoLa*⁺-shell users can manage the exploration state via two meta-commands inherited from the work in Chapter 2: `jump` and `revert`. With `jump`, users can jump to arbitrary states already seen during the exploration, which can be used to introduce branching. With `revert`, users can prune the exploration tree to throw away futile paths. In addition, users have access to the `mirror` meta-command, which results in mirroring of program execution across multiple branches. The `mirror` meta-command is fully defined in terms of `jump` and the execution of programs.

4.5.8 Domain Definitions

The main requirement for a domain definition is that its evaluation model must abide by the initial algebra semantics approach. To illustrate the extensibility this provides, we give an alternative semantic domain for the lambda calculus example. The example semantic domain we use to demonstrate is rendering operators as strings.

```
renderDomain :: Domain String
renderDomain = Domain "render" Render.parse renderAlg
```

```
renderAlg :: BuiltInOp String → String
renderAlg (OInt i) = show i
renderAlg (OString s) = s
renderAlg (OBool b) = show b
renderAlg (OTuple t) = show t
{- ... -}
```

With the `render` domain definition, we can give pretty printing semantics to operators.

```
pretty Var = @Var
pretty Abs = lambda @Var : @Body
pretty App = @Fun(@Arg)
```

The pretty printing example is rather simple. We can define a similar domain with some extensions to generate textual DOT programs for our DOT language defined earlier. With this domain, we map our operators directly to the corresponding concrete syntax of DOT. As a result, this domain cannot be applied to the for-loop extension we defined, but can be defined for the other operators.

```

dot Graph = "graph" spaced(@Name) '{' newline
    let edge = "--" in
        indent(@Component{'\n'}) newline
    '}'
dot Node = spaced(@Name) '[' @Annot{,} ']'
dot Edge = @Source $edge @Target
dot KV = @Key '=' @Value

```

This example boosts several operators to simplify the correct generation of a DOT program directly to concrete syntax. The `indent` operator ensures correct indentation, and the `@Operand{ c }` operator does a join of the sequence of values with the given character `c`. This highlights how an algebra can be adjusted to the domain, and this example algebra could be further developed as a more feature-full pretty printing algebra, possibly even in such a way that the for-loop extension introduced earlier is definable in the pretty printing domain. Note that designing algebras in a *generic* way is not trivial. For example, let bindings are used in the example to handle the way the our version of the DOT language models graphs and directed graphs, where the semantics of the edge depends on the type of the enclosing graph. This shows that there is an interplay between language design and algebra design.

4.6 DISCUSSION

In this section we discuss the results of our extended exact replication. We start by comparing *iCoLa* and *iCoLa*⁺ in §4.6.1. The remainder of the discussion focuses on the *iCoLa*⁺ specific parts.

4.6.1 *iCoLa* vs *iCoLa*⁺

Without looking at concrete syntax, we fully replicate the results obtained with *iCoLa*. This is explainable because the formal model implemented by *iCoLa*⁺ is an extension that fully encompasses the formal model implemented by *iCoLa*. However, when including concrete syntax in the comparison, we do see some decline in operator reuse as indicated in Table 4.1. Nevertheless, as Table 4.1 highlights, not all operators required a different concrete syntax definition. This indicates that the introduction of

concrete syntax does not nullify the reusability obtained with our approach. Furthermore, during exploration one might opt not to modify the concrete syntax definition of some operators but only do this after the choice of operators is finalized. Nevertheless, concrete syntax can have an impact on language ergonomics. In *iCoLa*, concrete syntax had to be defined separately from *iCoLa*, and is therefore not incremental, and was not integrated within the *iCoLa*-shell. With *iCoLa*⁺, concrete syntax is defined within *iCoLa*⁺, and integrated within environment definitions. Having concrete syntax be part of the language might improve explorations via the introduction of domain-specific syntax. It also removes the need to context-switch when working on parsers.

4.6.1.1 EDSL vs DSL

Since both *iCoLa*⁺ and *iCoLa* are built upon the same foundation, some of their main differences arise in the way users interact with the implementations. *iCoLa* provides an EDSL for interaction, while *iCoLa*⁺ provides a DSL. The benefits and limitations of this design choice are discussed in this section.

The DSL implementation provides concise definitions of languages and operators by removing boilerplate code. For example in operator definitions, the EDSL requires that the operand locations have type families and are matched in the constructor for the operator. In the DSL this is done automatically, as illustrated in Listing 4.4 by comparison of the `Abs` operator.

Definition of semantic functions is rather similar between the implementations, with the main difference that the DSL allows domains to define their own concrete syntax, resulting in more concise definitions. In our comparison this is mostly noted by the fact that the EDSL requires wrapping arguments in lists due to the variable number of arguments funcon constructors can take. In the DSL implementation, this is not needed.

Language definitions in the EDSL and DSL are also rather similar, with some differences in available syntax. For example in the EDSL, list comprehension can be used. Similar expressiveness is obtained with constraint sequencing in the DSL. The DSL also includes several forms of syntactic sugar that make language definitions slightly more concise.

Another benefit of the DSL is error reporting to the user. The EDSL was based on Template Haskell, and errors resulted in Haskell type errors. With the DSL implementation, we can give domain-specific errors instead and use the terms available in our language, such as `operator`, inside error message.

The DSL does not provide the full power of Haskell, making it more difficult for users to create abstractions, such as function composition or using `where` clauses to improve readability, and languages are not first-class citizens anymore. Difficulty in the creation of abstractions is clearly visible with refinements. In the EDSL, refinement functions can be composed, which is not possible in the DSL implementation because refinements are directly applied on languages instead of taking languages as parameters, resulting in some duplication. Not having languages be first-class citizens makes abstractions like parameterized languages very convoluted, which is shown in the DOT example, where the extension of the DOT language with directed graphs duplicated the earlier definition of DOT. With parameterized languages, a parametrized variant of the DOT language which is parametric in the graph could be defined, and a directed and undirected variant of the DOT language could then be defined using this parametric definition and unified into the language from our example

Some of the abstraction capabilities are still attainable via the extension points provided by *iCoLa*⁺. Nevertheless, in future work we would like to explore how to enable users to introduce abstractions natively in the DSL, or explore which abstractions are fruitful to include directly in the DSL. Examples of such abstractions are operator inheritance and parameterized refinements. With operator inheritance, operators can be built by composing other operators together and thus inheriting their semantic descriptions. With parameterized refinements, refinements can take a language as a parameter, making composition of refinements possible to some degree.

Implementation wise, the DSL required around two to three times as much code as the EDSL, with the remark that the tooling for the external DSL is almost non-existing. So far, we have implemented the *iCoLa*⁺-shell, similarly to the *iCoLa*-shell. However, no support for debugging, profiling, and editor services are

Listing 4.4: Comparison of the definition for the Abs operator in the EDSL (left) and the DSL (right).

```

data Abs u t where
  Abs :: IsTrue (AbsBody t)
        => String → u t → Abs u
        AbsType
type family AbsBody t
operator Abs : Var Body

```

available with the DSL, while the EDSL inherits this from the embedding in Haskell.

To summarize, *iCoLa*⁺ required a more substantial development effort compared to *iCoLa*. With the choice of a DSL, *iCoLa*⁺ sacrifices some expressiveness for more concise definitions, better error reporting and more flexibility. In addition, users can define operators and languages without any Haskell knowledge with *iCoLa*⁺, in contrast to *iCoLa* where Haskell knowledge is required.

4.6.2 Restrictions and Scalability

With the flexibility our approach provides, language definitions can become unwieldy where it is unclear where operators are exactly assigned to, which operators are part of the language, and how they are affected by specialization code. In our experience, the development is often done in layers, where prototyping is done at the current layer and when done, the layer is fixed. This keeps modifications local and prototyping focused on specific areas. It is important, however, that the first layer is well understood before such a development process can be applied.

Getting a layer correct is not trivial. This is shown in our definition of the DOT language, where edges have no annotations, which they do have in the real DOT language. Extending our language with such annotations is not easy and might require modifications to the both the semantics of the graph component and the semantics of the edge component. In certain situations, this can be solved with glue code, but that depends on the place in the semantics in which the modification is required. Certain

choices for the semantics of an operator are fundamental and can not be altered with glue code. This is a fundamental issue when making operators immutable, since there are limited places where glue code can be injected, namely at the places where operands are used and around the full operator. One way to solve this is by encoding the semantic domain itself in *iCoLa*⁺. The semantics domain then becomes the object language. Of course, this removes many of the advantages of *iCoLa*⁺ and the domain-specific abstraction introduced by an object language.

An alternative way to solve this is by enabling operations over operators, such as inheritance, as mentioned before. Inheritance of operators would enable new operators to be defined in terms of older operators, somewhat reminiscent of forwarding in attribute grammars [210]. An operation over an operator can also be defined that enables fine-grained modifications to the semantics by rewriting certain nodes in the semantics domain. However, enabling this in *iCoLa*⁺ is difficult due to the choice of allowing semantic domains to define their own syntax, which requires semantic domains to define a rewriting language to support this feature. A possible advantage of being less expressive for such situations is that when a modification is needed for a lower level, it provides an ideal opportunity to utilize the exploratory programming features of *iCoLa*⁺ by creating a new exploration branch where the modifications to a lower level are made. This keeps the different explorations more isolated compared to either modifying an existing operator in-place or introducing a variant of an operator with a different name and slightly different semantics.

In future work, we want to explore tooling that can help in quickly understanding the effects of new operator assignments and language composition, and supporting the aforementioned exploratory situations better. One way we aim to achieve this, is by utilizing visual views of the defined languages, as we have shown in Figure 4.1. One possible direction is to move away from text-based language engineering towards a visual style, by connecting operators with operand locations by drawing an edge between them, somewhat akin to graphical modeling languages. This allows a developer to see the structure of a language easier at a glance compared to a text based approach. How this scales to

larger languages and how it affects the ergonomics of a developer is something that needs to be investigated. Nevertheless, these extensions can be achieved in future work via environment definitions that manage the interaction. Consequently, these extensions require no modifications to the existing *iCoLa*⁺ implementation.

4.6.3 *Domain Definitions and Domain Fusion*

Through a standard example of a pretty printing extension, we have demonstrated that it is possible to add new domain definitions and modify the *iCoLa*⁺ DSL. With the possibility to add new domains, *iCoLa*⁺ can also be used as a vessel for the evaluation of new semantic specification languages. Currently, we are exploring defining domains for static semantics and scope graphs [149]. In future work, we will report on these efforts and investigate how to fuse domain definitions together. With domain fusion, translation functions can utilize other domains. Using information from another domain can result in more efficient specifications, for example, by utilizing typing information.

A weaker variant of domain fusion, but immediately useful in the context of *iCoLa*⁺, of domain fusion is sub-algebras. A sub-algebra works in the same domain as an algebra but only on a subset of the operators. This is useful when the semantics of an operator have multiple components. The for-loop operator in the DOT example is a prime illustration of this problem. The for-loop essentially requires four sub-algebras, which are now modeled using tuples, which makes the resulting translation functions more verbose and error-prone. Another example of this problem is local variables. In the case of funcons, local variables often use the scope funcon to load names in a local scope. When variable declarations and other operators can be mixed, one can implement this by splitting a variable declaration into two parts, the variable declaration and the assignment to the variable. The semantics of variable declarations can be split into two parts as well, the first being just the name of the variable, and the second being the assignment of the value to the variable. In this case, a sub-algebra could be defined that collects the name of all variables in a scope, which can then be passed to the scope

funcon in combination with a location to allow modifications to the variable.

Having access to sub-algebras would simplify the specification of more complex operators and improve the maintainability significantly. The main problem to include this in $iCoLa^+$ is how to denote a call to a sub-algebra in a translation function. Since translation functions can use their own syntax, a mechanism is required that allows a domain definition to communicate the needs for the result of a sub-algebra with $iCoLa^+$. We see two immediate ways to handle this. The first is to update the catamorphism to a paramorphism where we retain access to the term being transformed. Another option is to add a pre-process step that determines the sub-algebras required by a set of operators, and then adjust our definition of catamorphism to apply the catamorphisms of the sub-algebra first and enable access to these results by the main catamorphism. This setup does require that a sub-algebra has an implementation for every operator. For many sub-algebras this can be achieved via a default implementation for the sub-algebra. For example, for variables declarations the default implementation could be the empty map.

Another observed problem is the verbosity of translation functions in the current implementation. This is again visible in the DOT example, especially regarding the In operator. One way this can be solved is by introducing language-specific funcons, which capture some of the common aspects seen in translation functions for a certain language. Currently, this can be done via Haskell. It would be interesting to see if this is possible to include in the $iCoLa^+$ language itself. For example, via *macro* definitions or something similar. This would lower the barrier for introducing domain-specific abstractions within a semantic domain. However, this does raise the question of how to encode the application of such a macro in the semantic domain, and raises similar difficulties as with sub-algebras.

4.6.4 Language Composition and Syntax Ambiguity

Through the construction of several languages via composition, we have shown that our approach supports the extension, refinement and unification operators for semantic and syntax, as

introduced by Erdweg [65]. However, disambiguation can currently only be achieved by encoding the disambiguation rules inside the structure of the language, which is difficult to manage when composing languages and not always sufficient. As a result, our definitions of the *Imp*, *MiniJava*, and *SIMPLE* languages slightly deviate from the definitions present in the P_{LanCompS} case-study.

We are still exploring the best way to introduce disambiguation inside *iCoLa*⁺. The easiest approach is to extend the concrete syntax definition with many of the combinators for disambiguation available in the used GLL library. Alternatively, we can take a similar approach as used in the SDF formalism, where ordering on operators can be defined and associativity labels can be attached to operators. Another possible alternative is to use pattern matching for disambiguation [2]. Independent of these directions, we aim to investigate, and do this to a degree in Chapter 6, whether the debug friendliness of GLL parsing can aid disambiguation in an interactive style, such that it can be integrated into the design process.

4.6.5 Extensible Built-in Operators

In the current implementation we have decided upon a selection of built-in operators. This selection is based on the requirement we found during the construction of the several languages and fragments demonstrated in this chapter. The current selection will not be enough for all possible language definitions. Currently, addition of new built-in operators requires extension to the core of *iCoLa*⁺. Since built-in operators correspond to lexemes of a language, we expect that a better system for the addition of built-in operators is required. For now, we see two ways to achieve this. The first option is to add a catch-all built-in operator that bypasses the Haskell type system and then require domain definitions to handle these built-in operators. This approach is easy to implement in the current system and easy to understand from a users perspective. Since this approach bypasses the Haskell type system, it can result in run-time errors when a domain definition needs to handle an operator for which it has no handler. The second option is to use data-types à la

carte [199]. With data types à la carte, built-in operators can be defined independently and composed together. The composition is then given to *iCoLa*⁺ together with a lexing definition for the operators. Algebra definitions then become type-class instances which need to be implemented for all the built-in operators supported by the domain, which is checked by the Haskell compiler. This approach requires a bigger engineering effort to achieve in the current implementation, makes built-in operators mutable, and results in more complex operator definitions.

4.6.6 *Exploration Within iCoLa*⁺

iCoLa⁺ support exploratory programming out of the box via its design as a sequential language and embedding within existing tooling. Exploring multiple ideas via the creation of many variants and discarding futile paths is fully supported via the jump and revert meta-commands present in *iCoLa*⁺. Although we currently only provide an exploratory REPL, in future work we want to investigate alternative interfaces with an explicit focus on exploratory language development, which can be defined as environment definitions. Towards this idea, we also aim to utilize the presented implementation to investigate exploratory patterns within language development to further guide interface design. Furthermore, within the *iCoLa*⁺-shell, a user can commit to a language and experiment with the object language within an object REPL. However, after exiting the object REPL, the object REPL session is lost. But, within an exploration session, a user might want to compare two languages by their usage within the object REPL. Currently, that is possible by scrolling back in the REPL, but there is no real support for it. In future work, we aim to investigate how to support retention of the object REPL sessions within the exploration session in an ergonomic and efficient manner.

4.6.7 *Limitations and Threats to Validity*

The primary evaluation of our approach is based on the semantic domain which uses funcons. This limits the scope to the class of languages which can have their semantics expressed in funcons.

Since the funcon library is open-ended [29], this class is mostly characterized by the fixed set of semantic entities. Nevertheless, a variety of languages already have their semantics expressed in funcon terms [31, 145]. Furthermore, our approach is extensible through new semantic domains. An interesting foundation for an alternative semantic domain is algebraic effects and handlers [166, 167], which provide a mathematical approach for reasoning about effects in programming languages and support composition [35, 175].

The usage of catamorphisms to guide the translation from the initial algebra to semantic algebras constitutes a fundamental functionality to our approach. Nevertheless, the initial algebra semantics presents a unified approach to formal semantics of programming languages [76], and therefore supports different approaches. However, this choice of abstraction puts certain restrictions on the translation functions used in our approach, which can affect the manner in which semantic domains are defined and used.

4.7 RELATED WORK

Developing languages via some form of composition is supported by a wide variety of language-development environments [66, 96, 201, 204, 213]. Erdweg et al. [65], performed a systematic evaluation of existing environments and their support for the different forms composition (extension and unification). Out of the considered environments, only JastAdd [63], which is an environment for the construction of Java like languages, supported unification at the semantic level. For syntax, both Spoofox [96] and SugarJ [66] support unification. To handle ambiguous grammars, Spoofox and SugarJ use the SDF formalism [83]. Our approach supports unification at both the syntax and semantic level, with the remark that disambiguation at the syntax level is minimal. Much of the syntax available in concrete syntax definitions of *iCoLa*⁺ are influenced by SDF. In contrast to SDF, we use GLL parsing instead of scannerless Generalized LR parsing.

Bertolotti et al. [20] extend Erdweg's framework by introducing six forms of reusability focused on different levels of granularity. With *iCoLa*⁺ we support the following forms: *sub-language compo-*

sition via language composition; *language feature composition*, since in $iCoLa^+$ every language feature — a combination of syntax and semantics — is a language and therefore can be composed via language composition; *syntactic and semantic assets composition*, since this is also modeled using languages in $iCoLa^+$; and *action extension* is partially supported via glue code. We do not support: *semantic assets composition*, since we do not allow usage of algebras inside an algebra definition; and *action composition*, since we do support composition of algebras. The two forms we do not support align with our earlier discussion on sub-algebras and domain fusion. Full support for action extension would be achieved with support for operations over operators, such as inheritance.

Lisa [135] is a full-fledged interactive environment for programming language development based on attribute grammars with support for incremental language development [136] via multiple attribute grammar inheritance [134]. Lisa also supports visual based development of programming languages. Compared to our approach, no distinction between operators and where operators are used is made.

Melange [59] is a meta-language involving meta-models and aspect oriented programming. It uses aspects to implement the semantics of languages, and supports both extension and unification. Our operator specialization closely resembles the idea of aspects as seen in Melange. Compared to our approach, Melange makes no distinction between operator semantics and operator specialization; does not make the distinction between operator definitions and operator usage; and operators are not immutable, instead a renaming mechanism is provided to solve conflicting abstract syntax. Multi-level modeling [12] supports more than two meta-modeling levels and has been used in language development to achieve extensible meta-models via linguistic extensions [112] and by specializing meta-models to specific domains via instantiation [116]. However, to support optionality of language primitives (closed variability), multi-level modeling needs to be combined with the product lines approach [113]. Within our approach, such optionality is achieved via refinements on language definitions. Perspectives [4] are a layer above the model layer and are used to describe the relations between multiple

languages and consistency requirements among them, or to exclude language concepts from the model layer. The approach has similar characteristics as Melange, with the addition that it enforces consistency requirements among languages. Our approach explicitly has few restrictions to promote the exploration phase. However, outside the exploration phase, more refined restrictions might be beneficial. There seems to be a parallel here with dynamic and static typing, where our approach is more related to dynamic typing. This seems to imply that a strategy alike to gradual typing is an interesting avenue to explore.

In Feature-oriented programming [7], a system is decomposed in the features it provides. This style of programming aims to increase structure, reuse and variation by making features user configurable such that a system can be developed by picking and configuring the correct features. Neverlang [38] is a Java-based development environment with support for language unification, modeled around the idea of feature-oriented programming. Neverlang uses evaluation phases for semantic specifications and supports dependent evaluation phases, while *iCoLa*⁺ does not. Evaluation phases and concrete syntax are combined into slices, which capture a language feature in isolation. A language definition then combines slices and determines the order of the evaluation phases. Although Neverlang does not enforce uniqueness of operand locations, it does provide some capabilities to remove structural choices via renaming. With renaming, a non-terminal can be renamed to another non-terminal at the language level.

Software product lines [43] is a development paradigm that models the software development process as a product line, where a system is constructed by selecting components from a repository, adapting the components to the use case, and integrating the components together. Compared to feature-oriented programming, software product lines focus on similarities between systems, also known as families. This gives a high variability where variants of systems can be quickly created. Feature-oriented programming can be used to implement software product lines, which is done by AiDE [110]. AiDE provides an environment for language-development based on software product lines by building an environment on top of Neverlang [38]. Besides

AiDE, there are several other environments integrating software product lines in the context of language development — also known as language product lines [131].

A focus on language families [123], a set of related languages, is inherent in the language product lines style of development. As a result, the variability of these systems is high, enabling the construction of a wide variety of languages in an incremental manner. However, because the focus is on language families, there is a restriction on the structure of the different variations. Nevertheless, correctness of model properties can be efficiently checked, which opens the door to promote the variability offered by product lines to more areas such as model editors and code generators [80]. Language product lines have been combined with multi-level modeling at the (abstract) syntax level [114] to enable both extension and selection based on a feature model. The approach supports bottom-up extensions where a meta-model is extended from below, which can be useful during the exploration process. To also enable modularity at the semantic level, graph transformations have been used [115]. With graph transformations, consistency of semantic constraints can be enforced among the languages within a language family. Our approach achieves modularity at the semantic level by supporting the introduction of new semantic domains and the introduction of new operators accompanied by semantic translation functions. Although our approach essentially describes a constraint graph, as indicated by Figure 4.1, we have not yet explored whether this can be utilized to enforce constraints without affecting the exploration capabilities, or aid the exploration process.

Concern-oriented language development [44] moves away from the family constraint by combining different modularity approaches at the language development level via so called concerns: reusable piece of language artifacts. Concerns have a variation interface, a customization interface, and a usage interface. The variation interface represents configurable components and the customization interface describes how a concern can be integrated into a different context. The exploratory capabilities of a concern are thus determined by the flexibility of these interfaces and the inherit restrictions present in concern definitions. The ideas of concern-oriented language development are used to

reuse language components that are textual, external, and translational [36]. The approach uses specific composition operator to ensure compatibility within the used technologies, which makes it impossible to remove parts of a language.

The usage of catamorphisms as an implementation technique is not unique, and for example also used in the context of attribute grammars. From the perspective of attribute grammars [71, 88], the algebras in $iCoLa^+$ only deal with synthesized attributes and the translations from operators to semantics functions can be seen as describing an S-attributed grammar. Supporting inherited attributes can be achieved via the semantic domain itself. For example, the semantic domain for concrete DOT programs uses an *indent* construct, which essentially encodes a inherited attribute for the indentation level. The literature in this area provides useful implementation techniques for extension to $iCoLa^+$, such as sub-algebras, which we aim to explore in future work.

4.8 CONCLUSION

In this chapter we introduced $iCoLa^+$, an extensible meta-language aimed at improving the language design process via rapid prototyping with reusable components and exploratory programming. $iCoLa^+$ extends the $iCoLa$ meta-language by adding support for concrete syntax, by providing a DSL for language definitions, and by supporting an arbitrary amount of semantic domains. The $iCoLa^+$ implementation is extensible via Haskell-defined environment definitions and domain definitions. Environment definitions determine how users interact with $iCoLa^+$ and the defined languages. Domain definitions determine the capabilities of semantic translation functions.

By constructing several languages with our approach, we have demonstrated to which extent our approach simplifies the construction of new languages as well as variants of existing languages. Through the construction of the $iCoLa^+$ -shell and by adding a new domain definition, we have shown the possibilities of extending $iCoLa^+$. The flexibility provided by $iCoLa^+$ makes it easy to modify existing language design choices, but also increases the difficulty of tracking the precise composition of languages when applied at (large) scale. In addition, disambiguation

of concrete syntax is only supported in a limited form. Methods to improve *iCoLa*⁺ in these regards are to be explored in future work.

ON THE SOUNDNESS OF AUTO-COMPLETION SERVICES FOR DYNAMICALLY TYPED LANGUAGES

In the previous chapters we have mainly looked at the meta-side of language engineering. In this chapter we shift our focus more towards the object language by looking at editor tooling used by developers during development. Tooling makes developers more productive at certain tasks, such as writing code and finding definitions. Having access to such tooling during the prototyping phase can aid evaluation of an object language variant by making the construction of substantial programs more feasible. The focus in this chapter is on dynamically typed language. This is motivated by our usage of funcons as a semantic domain throughout this thesis, which is a dynamically typed language. If we obtain tooling for funcons, we can also obtain tooling for the languages defined in terms of funcons. Hence, in this chapter we look at auto-completion services for dynamically typed languages, with a primary focus on the soundness of such tooling.

Associated Publications

- **Damian Frölich** and L. Thomas van Binsbergen. “On the Soundness of Auto-completion Services for Dynamically Typed Languages.” In: *Proceedings of the 23rd ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. GPCE '24. Pasadena, CA, USA: Association for Computing Machinery, 2024, pp. 107–120. ISBN: 9798400712111. DOI: 10.1145/3689484.3690734

5.1 INTRODUCTION

Today’s developers have access to an abundance of programming tools that increase their productivity by assisting the program-

ming activity itself or the management of (large) code bases through, for example, version control or delivering project insights. Integrated Development Environments (IDEs) integrate such tools and serve as the main interface for programmers during programming-related activities. Today, most IDEs are decoupled from language implementations or programming analysis tools such that a single IDE can be used for multiple languages and multiple IDEs can be used to work on the same code base (e.g., chosen based on programmer-preference). The Language Server Protocol¹ (LSP) plays a crucial role in this decoupling, interfacing between the environment and so-called *language services* such as ‘go-to definition’ and auto-complete. As discussed in Chapter 2 where we introduced the Exploratory Programming Protocol (EPP), the client-server architecture of the LSP addresses the $n \times m$ problem of supporting n languages in m IDEs, reducing the (overall) engineering efforts for both IDEs and languages. Moreover, by modularizing the server back-end into distinct services, tools can be developed that could specialize in particular program analyses or programmer feedback.

Language services typically analyze source code. The type of service determines the density of information to be extracted. For example, an abstract syntax tree provides enough information to implement semantic highlighting (an extension of syntax highlighting). Other services may require additional information, such as the types of variables or the declaration to which a reference resolves. For statically typed languages, the structure enforced by type systems aid the implementation of such services. Inherent to statically typed languages is a static type-checker, which can catch many coding inconsistencies before a program runs. In contrast, a significantly more complex analysis is needed to perform (static) type-checking for dynamically typed languages as the type of a variable may be determined by program input [141]. More generally, editor services for dynamically typed languages may require complex analyses which may not always capture all cases, or make a trade-off between soundness and completeness.

The state of the art in auto-complete services for Python exhibit this phenomenon, with tools generally choosing completeness over soundness. For example, an auto-complete service provid-

¹ <https://microsoft.github.io/language-server-protocol/>

Listing 5.1: Python example on which completions from auto-completion services can introduce erroneous execution paths. The ? character indicates the position of the cursor, i.e. the source location from which an auto-complete request is performed.

```

class A:      obj = A()
    x = 5      obj.x = input
    z = 10     ()

class B:      if obj.x:
    x = 0      obj = B()
    y = 5

                print(obj.?)

```

ing auto-completion candidates for the program in Figure 5.1 at the position indicated by the question mark, needs to take into account that the `obj` variable can point to an object of class A or to an object of class B, depending on user-provided input. As a result, the completion candidates `y` and `z` introduce an erroneous execution path since they are not present in both A and B. The `x` field is present in both classes and is therefore a sound completion candidate, since in both run-time paths that field is present on the object assigned to the `obj` variable. However, Pylance² and Jedi³, two prominent editor service implementations for Python, give all three fields as completion candidates. This result is complete as it contains all candidates that create a valid program flow but is not sound as some candidates introduce erroneous program flows (with respect to name resolution).

With small programs, the programmer may be able to detect unsound candidates and handle them appropriately. However, for more complex programs, maintaining the oversight required to do so becomes challenging. Furthermore, the execution of some erroneous execution paths may be rare, complicating bug discovery. Obtaining such erroneous completion candidates is

² <https://github.com/microsoft/pylance-release>

³ <https://jedi.readthedocs.io/en/latest/>

mentioned as one of the most concerning issues by practitioners when using auto-completion tools [218].

In this chapter, we introduce an approach that sets a first step towards the creation of sound editor services for dynamically typed languages. The approach leverages abstract interpretation and scope graphs to build a model for resolving names in programs. Using this model, we can construct auto-completion services that are sound with respect to the name binding of a program across the different execution paths the program embeds. Concretely, in this chapter we make the following contributions:

- an approach based on abstract interpretation and scope graphs for the implementation of sound completion services;
- an implementation of said approach in Haskell;
- a test set of Python programs with key name binding challenges that result in unsound completion candidates with the state of the art editor services.

This chapter is structured as follows. In §5.2 we give the necessary background. In §5.3 we discuss the difficulties of applying scope graphs to dynamically typed languages, and present our extension to the scope graph framework. We follow this up by obtaining scope graphs from the run-time heap in §5.4, and via abstract interpretation in §5.5. In §5.6 we introduce an implementation of our approach, and demonstrate it in §5.7 via a comparison with the state of the art editor services for Python. We discuss the results from our experiments and our approach in §5.8. We finalize with related work in §5.9, and our conclusion in §5.10.

5.2 BACKGROUND

Our approach utilizes abstract interpretation to obtain a sound over-approximation of the name binding seen at run-time. The name binding is captured using scope graphs. These two components combined form the basis for an auto-completion service implementation.

5.2.1 Abstract Interpretation

Abstract interpretation [48] provides a unified framework for sound static analysis by over-approximation of the dynamic semantics of a programming language. With abstract interpretation, the concrete domain is approximated by an abstract domain. The abstract domain has less computational needs. However, since it is an over-approximation, some information is lost, affecting completeness. The concrete and abstract domain are related via a Galois connection, with which values from two different partially ordered sets can be related. A Galois connection on two partially ordered sets (C, \leq_C) and (A, \leq_A) is given by two monotone functions $\alpha : C \rightarrow A$ and $\gamma : A \rightarrow C$, such that $\alpha(c) \leq_a a \iff c \leq_c \gamma(a)$ holds for all $c \in C$ and $a \in A$. Intuitively, using α we can relate concrete elements to abstract elements, and using γ we can relate abstract elements to concrete elements, and it is ensured that the α and γ functions respect a specific relation between these concrete and abstract elements. In the context of abstract interpretation, this relation is often seen as the degree of information one can extract out of a value. The *higher* an element is in the partial order, the less information one is able to extract out of it. The Galois connection then captures the idea that one can not use the α or γ function to obtain an element with more degree of information from an element with a lesser degree. Instead, the degree of information might be the same or reduces, but never increases. The functions (α, γ) are thus monotonic on this degree of information.

To make this concrete, we illustrate the idea using the abstract domains of signs of integers, which we encoded as a DOT program in the previous chapter. In this example, the concrete domain are sets of integers $(\mathcal{P}(\mathbb{Z}))$ and the abstract domain is the set of signs: $P = \{\perp, <0, Z, >0, \top\}$. The abstraction function maps a set of integers to a (shared) sign.

$$\begin{array}{ll}
 \alpha(\emptyset) = \perp & \alpha(\{0\}) = Z \\
 \alpha(\{i \mid i < 0\}) = <0 & \alpha(\{i \mid i > 0\}) = >0 \\
 & \alpha(\mathbb{Z}) = \top
 \end{array}$$

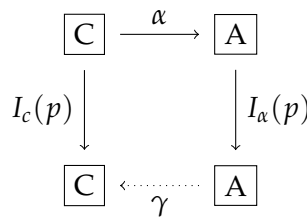
The concretization function can be mechanically derived from the abstraction function and the requirements of a Galois connection. In this example, obtaining the concretization function is simply just the mapping of the α function reversed:

$$\gamma(>0) = \{i \mid i \in \mathbb{Z} \wedge i > 0\}.$$

Alternatively, one can define the concretization function and derive the abstraction function, or simply define both without deriving. The advantage of deriving via the Galois connection is that the Galois connection then holds by definition.

Our concrete domain together with the subset relation form a partial order. We can thus demonstrate that for all sets of integers, the Galois connection requirement holds. For instance, if we start with the set $\{-1, 0, 1\}$ and apply the abstraction function, we obtain \top . If we concretize this, we obtain \mathbb{Z} , and indeed $\{-1, 0, 1\} \subseteq \mathbb{Z}$ holds. Alternatively, if we start with $\{1, 2, 3\}$ and abstract this, we get >0 . If we concretize this, we obtain \mathbb{Z}^+ , and indeed $\{1, 2, 3\} \subseteq \mathbb{Z}^+$ holds. In the first case, the property of the final result only tells us that the original set contained integers, but does not tell us more. In the second case, the result tells us that the original set only contained positive integers. Sets thus capture properties.

Using the Galois connection and the abstract domain, we can do static analysis on programs by interpreting the programs in the abstract domain. The advantage of this is that interpreting the program in the abstract domain generally takes less computational power. For example, a program might not terminate in the concrete domain while it will terminate in the abstract domain. The following diagram sketches the idea behind abstract interpretation and the usage of the Galois connection, where the dotted line denotes an over-approximation.



Essentially, for every program we can interpret the program ($I_c(p)$) on a concrete configuration (c) and obtain a new concrete configuration. Or, we can abstract the current configuration, execute the program using the abstract interpreter ($I_\alpha(p)$) and concretize the resulting abstract configuration, giving an over-approximation of the configuration obtained from interpreting the program in the concrete domain. This over-approximation captures a property of the executed program, and since the concrete configuration is contained in this over-approximation, the concrete configuration also has the property. This thus gives us a way to capture properties of programs via the interpretation in the abstract domain. This diagram is not achieved for every program by every choice of the abstract interpreter. Instead, both interpreters must be monotonic and the abstract interpreter must adhere to the semantics of the interpreter in the concrete domain, such that the diagram holds. Again, this can be achieved by deriving the abstract interpreter from the concrete interpreter and the diagram. Notionally, we require that for every program p and configuration c , the following holds $I(p, c) \sqsubseteq \gamma(I_\alpha(p, \alpha(c)))$.

We can extend our earlier example of sets of integers with programs capturing integer addition. The concrete evaluation is then the standard evaluation of integer addition expressions. The abstract evaluation becomes the evaluation of addition on signs. For example, adding zero to a positive number results in a positive number: $Z + >0 = >0$, and when adding a positive number to a negative number the resulting sign is unknown: $<0 + >0 = \top$. In the second case, we do not know the sign, and the result can be positive, negative, or zero, thus the operation yields \top (representing the set of all integers). Furthermore, these operations are strict on \perp elements, e.g., $\perp + Z = \perp$ (with \perp representing divergent computations). In our example, the property captured by our abstract interpreter is the sign of an integer addition expression. This information can be used, for example, by a compiler to optimize conditionals that are always true or false.

Although the example is trivial and the computation in the abstract domain does not differ that much from the concrete domain, there are still situations where the abstract domain does significant less work. For example, any expression without a \perp

element but containing a \top element in it evaluates to \top . Hence as soon as a \top element is obtained, evaluating the remainder of an expression is futile, under the assumption that no \perp element exists in the expression. Of course, getting \top as a result gives minimal information about the properties of the program. By picking a more precise abstract domain, we might obtain more information from the abstract evaluation while likely increasing the computational needs of the abstract interpreter. There is thus a trade-off between the preciseness of the abstract domain and the amortized time it takes to evaluate a program in the abstract domain. The most precise abstract domain is simply the concrete domain.

ABSTRACT INTERPRETATION AND NATURAL SEMANTICS

Abstract interpretation has been applied by Schmidt [188, 190] to languages with an operational semantics in the small-step style (of Plotkin [165]) or big-step style (of Kahn [92]). In the approach, concrete and abstract computations are related by safety relations on values and trees. Every computation represented as a concrete tree needs to be mirrored by an abstract tree and the safety relation on trees needs to be preserved. When a safety relation (*safe*) is both U-closed ($c \text{ safe } a \wedge a \sqsubseteq a' \implies c \text{ safe } a'$) and G-closed ($c \text{ safe } \sqcap \{a' \mid c \text{ safe } a'\}$), a Galois connection is obtained for free. For our sign example, we would thus define a safety relation between integers and signs.

$$\begin{aligned} i = 0 &\vdash i \text{ safeVal } Z \\ i < 0 &\vdash i \text{ safeVal } <0 \\ i > 0 &\vdash i \text{ safeVal } >0 \end{aligned}$$

Note that the safety relation makes no mention of the \perp and \top elements. Nevertheless, the relation between \emptyset and \perp , and \mathbb{Z} and \top still holds. To illustrate this, we can define α in terms of the *safeVal* relation as follows: $\alpha(C) = \sqcup \{a \mid c \text{ safeVal } a, c \in C\}$. In case $C = \emptyset$, we get $\sqcup \emptyset = \perp$, which aligns with our definition of α from before. When C contains values with different signs, for example $C = \{-1, 0, 1\}$, we get $\alpha(C) = \sqcup \{<0, Z, >0\} = \top$ by our definition of the lattice of signs. This relational approach

to defining Galois connections via safety relations is what we use in this chapter for our definitions of abstract domains.

5.2.2 Scope Graphs

Scope graphs have been introduced by Neron et al. [149] to capture name binding in a language-independent manner. In this paper we adopt the formal definition of scope graphs given in Figure 5.2. The vertices of a scope graph represent scopes, references, or declarations, and are assumed to be unique such that each declaration and reference belongs to one scope. References and declarations are indexed to distinguish occurrences with the same name at different source locations. When the index is clear from the context, we omit it. Edges denote the relation between scoping elements. A reference has an edge to the scope in which it occurs ($x_i^R \rightarrow s$). References are identified with a superscript R . Scopes have edges towards the declarations made within that scope ($s \rightarrow x_i^D$). Declarations are identified with a superscript D . Both references and declarations have a subscript that resembles the location in the source code and is used to differentiate between different occurrences of the same identifier. Scopes can also have labeled edges towards other scopes ($s \rightarrow_l s'$). This can be used to model, for example, lexically nested scopes. Figure 5.1 shows an example scope graph for a very simple program in a functional language with imports. The program calls the function f . The function f is defined in the module M . To find the declaration associated with f when called, we start from the reference to f in scope s_2 and move upward to s_1 via the parent edge, then we move upward again via the parent edge to s_0 . From s_0 we follow the import edge to m_1 , which is the main scope of the module M . From m_1 there is an edge to the declaration of f .

A declaration can also have an edge towards a scope when it introduces this scope ($x_i^D \rightarrow s$). For example, the scope of an object containing the fields and methods of that object. An object access ($x.y$) is then translated into resolving the variable x and determining whether the resulting declaration has an outgoing scope s . The variable y is then resolved starting from scope s . In this paper, as well as in [149], labels **P** (parent) and **I** (import) are used. However, the usage of imports in our work is more

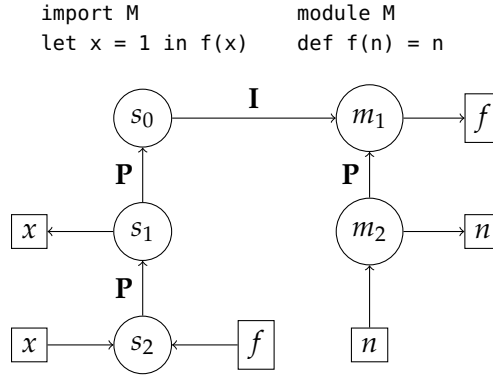


Figure 5.1: The scope graph for a simple program in a functional language with imports.

rudimentary than that of [149]. We also assume that every scope has at most one parent (i.e., one outgoing edge labeled **P**).

Cycles are permitted in scope graphs. To resolve a reference to a declaration, a resolution calculus is defined that describes correct resolution paths in a scope graph, given by the \mapsto relation in Figure 5.2. The resolution calculus keeps track of a set of seen scopes and only visits a scope once. A valid path in a scope graph is a sequence of edges moving from the current scope to scope s via the edge with label l , denoted by $\mathbf{E}(l, s)$; and declarations reachable from the current scope, denoted by $\mathbf{D}(x_i^D)$. The resolution in our example from the reference of f to its declaration is captured by the following path: $\mathbf{E}(\mathbf{P}, s_1) \cdot \mathbf{E}(\mathbf{P}, s_0) \cdot \mathbf{E}(\mathbf{I}, m_1) \cdot \mathbf{D}(f)$. Using the resolution calculus and a language-specific ordering on paths, we can define the visible declarations in a scope by finding all shortest resolution paths reachable from the scope, which is defined in Figure 5.2 by the \rightarrow arrow, i.e. $s \rightarrow (s', x_j^D)$ states that from scope s the declaration x_j^D in scope s' is visible (not shadowed).

A reference can resolve to multiple declarations by finding multiple resolution paths. Language-specific path orderings and well-formedness predicates are used to determine the desired resolution. In the formal model this is reflected in the $(ResE)$ rule and the (Vis) rule. The ordering selects the ‘nearest’ declaration site. Different name binding policies can be modeled with the well-formedness predicate [5]. The chosen policies may still not

Scope graph

$$\begin{aligned}
s &\in \text{ScopeId} \\
v \in \text{Vertex} &::= s \mid x_i^D \mid x_i^R \\
e \in \text{Edge} &::= s \rightarrow_l s \mid s \rightarrow x_i^D \mid x_i^R \rightarrow s \mid x_i^D \rightarrow s \\
l \in \text{Label} &::= \mathbf{P} \mid \mathbf{I} \\
G \in \text{ScopeGraph} &::= \mathcal{P}(\text{Vertex}) \times \mathcal{P}(\text{Edge})
\end{aligned}$$

Projection functions

$$\begin{aligned}
K(s) &= \{l \mapsto \{s' \mid s \rightarrow_l s'\}\} \\
D(s) &= \{x_i^D \mid s \rightarrow x_i^D\} \\
R(s) &= \{x_i^R \mid x_i^R \rightarrow s\}
\end{aligned}$$

Resolution paths

$$p \in \text{Path} ::= \mathbf{D}(x_i^D) \mid \mathbf{E}(l, s) \cdot p$$

Paths

$$\begin{array}{c}
\frac{\vdash_G s \rightarrow x_i^D}{\vdash_G \mathbf{D}(x_i^D) : s \mapsto (s, x_i^D)} \quad (\text{RESD}) \\
\\
\frac{s' \notin S \quad S \vdash_G s \rightarrow_l s' \quad \{s'\} \cup S \vdash_G p : s' \mapsto (s'', x_i^D)}{\vdash_G \mathbf{E}(l, s) \cdot p : s \mapsto (s'', x_i^D)} \quad (\text{RESE}) \\
\\
\frac{\vdash_G x_i^R \rightarrow s \quad \{s\} \vdash_G p : s \mapsto (s', x_i^D)}{\vdash_G p : x_i^R \mapsto (s', x_i^D)} \quad (\text{RESR})
\end{array}$$

Visible declarations

$$\frac{\{s\} \vdash_G p : s \mapsto (s', x_i^D) \quad \forall j, p', s'' (\{s\} \vdash_G p' : s \mapsto (s'', x_j^D) \implies p' \not\prec p)}{\vdash_G p : s \mapsto (s', x_i^D)} \quad (\text{VIS})$$

Figure 5.2: Formal definition of scope graphs with resolution calculus, and parameterized by a well-formedness predicate over paths and an ordering on paths. Based on earlier definitions [6, 170]

select a unique resolution, for example, when the program is invalid. The scope graph framework leaves it to the user to determine whether this is (un)desirable based on the context in which the framework is used.

Prior work has identified a correspondence between static name binding in scope graphs and heap-allocated frames [170]. In essence, a scope graph functions as a blueprint for the heap at run-time. The correspondence brings several benefits, such as uniform type soundness proofs and sound garbage collection. In this work, we utilize this correspondence in the other direction: we use the heap and frames approach as a blueprint for building scope graphs.

5.3 SCOPE GRAPHS FOR DYNAMIC LANGUAGES

Dynamic languages may be dynamic in several regards, e.g., name resolution may be dynamic, the types of variables may be established dynamically, or programs may be extended dynamically. Similarly, at least three types of correctness can be identified: name binding correctness, type correctness, and syntactic correctness. In this paper, we are primarily interested in the first type of correctness and consider an auto-completion candidate to be sound when its inclusion yields no name resolution errors at run-time. In this section, we describe how we annotate and build scope graphs to model dynamic name resolution, a first step in our approach towards sound (auto-completion) services for dynamically typed languages.

Scope graphs have been used to implement sound auto-complete services for statically typed languages [158]. We observed two main problems when applying scope graphs to dynamically typed languages, which we shall demonstrate by comparing the scope graph of a small statically typed (functional) program and the scope graph of a dynamically typed program (following the name binding semantics of Python). Figure 5.3 contains the example programs and the corresponding scope graphs. In both programs, some dynamic input is assigned to the variable x and is used for branching into one of two conditional branches, which both introduce a binding. (The subsequent code in the branches is irrelevant to our example.) Both programs have a

'global' scope labeled s_0 (in the graph and program text) and evaluate the dynamic input in this scope before it is assigned to x . The assignment to x creates a new scope (s_1) in case of the statically typed program. For the dynamically typed program, it adds a declaration to the global scope. Based on the value of the x variable, one of the two bodies is executed. In both bodies, an assignment is performed. In case of the statically typed program, the right sides of the assignments are evaluated in scope s_1 , and for both declarations a new scope is created s_2 and s_3 , respectively. For the dynamically typed program, a new scope is created for the `if` body (s_2) and for the `else` body (s_3), in which the respective bodies are executed. Both scopes have a parent edge to the global scope. The global scope also has an import edge to both scopes. The import edge is required because after the `if-else` we are back in the global scope (s_0), but the declarations made in the body are still reachable, which is modeled using the import edges. In our statically typed program, this is not the case.

From a resolution perspective, we could have given a variety of different scope graphs for the dynamically typed program that gives the same resolution results. For example, one scope containing all declarations. Why we have opted to display the scope graph for the dynamically typed program as having multiple scopes, becomes apparent shortly.

The first problem is that a dynamically typed language puts less restrictions a-priori on a program, which makes the resulting scope graph an over-approximation of the name binding seen at run-time. As a result, reasoning with the resulting scope graph requires care. In our example, this is illustrated by the fact that in both programs, the declarations y and z are never in scope at the same time. This is captured in the scope graph obtained from the statically typed program. However, the scope graph for the dynamically typed language does model that both declarations are in scope: we can take an import edge from s_0 to both s_2 and s_3 and obtain the respective declarations. This is incorrect. The dynamically typed program actually has two scope graphs. One in which there is an import edge towards scope s_2 , and thus y is declared; and one in which there is an import edge towards scope s_3 , and thus z is declared.

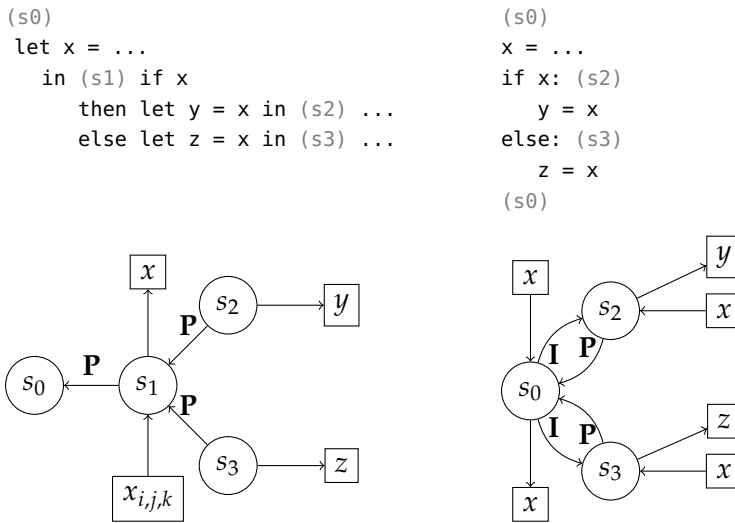


Figure 5.3: A similar program written in a statically typed (functional) language and a dynamically typed language with Python like scoping rules, with their corresponding scope graphs. Scope changes in the programs are indicated with parentheses. i, j, k denote source locations for the different references to x in scope s_1 .

The second problem is that the scope graph for the dynamically typed program only describes the name binding at a specific program point. As a result, the obtained scope graph cannot be used to reason about every program point. In our example this is illustrated if we use the obtained scope graphs and query the available declarations in the conditional of the `if-else`. In the statically typed scope graph, the scope in which the conditional is executed is s_1 , so we start our resolution from that scope, and obtain only x as a declaration. For the dynamically typed language, the condition expression is executed in scope s_0 , and from that scope we obtain the declarations x, y, z , where y and z are obtained via the import edges. However, at that point, only x is in scope. Our scope graph for the dynamically typed language is thus only valid at the end of our program, but not at intermediate stages.

5.3.1 Annotated Scope Graphs

To solve the first problem, we extend the scope graph framework with annotated vertices to denote uncertainty.

As observed, a scope graph for a dynamically typed language describes an over-approximation of the scope graphs seen at run-time. In our example, the obtained over-approximation only describes which variables *might* be in scope. However, it is unknown which variables are *definitely* in scope. Reasoning with our example scope graph results in unsound completion candidates. To overcome this, we extend the scope graph definition with annotations on declarations and references. We define the set of annotations as $An = \{N, D, M\}$, representing not present, definitely present, and maybe present, respectively. The annotation set forms a join-semilattice by the following partial order $N \leq_{An} M$ and $D \leq_{An} M$. The vertices of the scope graph model are updated with an annotation component: $v \in Vertex ::= s \mid (x_i^D, a) \mid (x_i^R, a)$ where $a \in An$. We also add an annotation to labels on edges, which models the uncertainty of edges between scopes. We do not annotate scopes, because we operate from a scope perspective: the correct annotation for a scope differs depending on the scope from which we observe the annotated scope.

Using the annotated scope graph model, we define a partial order on scope graphs as follows, where D_v projects the annotated vertices with annotation D , and D_e projects the labeled edges with annotation D .

$$\begin{aligned}
 (V_1, E_1) \leq_s (V_2, E_2) \text{ iff} \\
 & \forall v \in V_1. \exists v' \in V_2. v \leq_v v' \\
 & D_v(V_2) \subseteq D_v(V_1) \\
 & \forall e \in E_1. \exists e' \in E_2. e \leq_e e' \\
 & D_e(E_2) \subseteq D_e(E_1)
 \end{aligned}$$

The order on vertices (\leq_v) and edges (\leq_e) is defined by equality on vertices and labels, and by comparison of the annotations using the \leq_{An} relation. A small excerpt of the full definition is given by the following two cases:

$$\begin{aligned}
 s \leq_v s' \text{ iff } s = s' \\
 (v, a) \leq_v (v', b) \text{ iff } v = v' \wedge a \leq_{An} b.
 \end{aligned}$$

We lift this partial order to a lattice (\sqcup_s) by adding a bottom and top element. The partial order on scope graphs captures the idea of over-approximation as introduced in the previous section. The second and last constraints ensure that the definitely present vertices and edges in the second scope graph are also present in the first graph. This ensures that conclusions made over the definitely present vertices and edges in the ‘abstract’ scope graph also hold in the concrete scope graph.

Figure 5.4 displays the two scope graphs observed in our dynamically typed example program and the annotated scope graph that over-approximates both. In our annotated scope graph, all the declaration are annotated as definitely present. But, the two import labels are annotated as maybe present. Performing a query from s_0 will not follow the maybe present edges, and thus will not obtain the declarations y and z , only the declaration x . When performing a query from another scope, for example s_2 , we will collect both y and x as definitely present declarations. From the perspective of scope s_2 , this indeed holds.

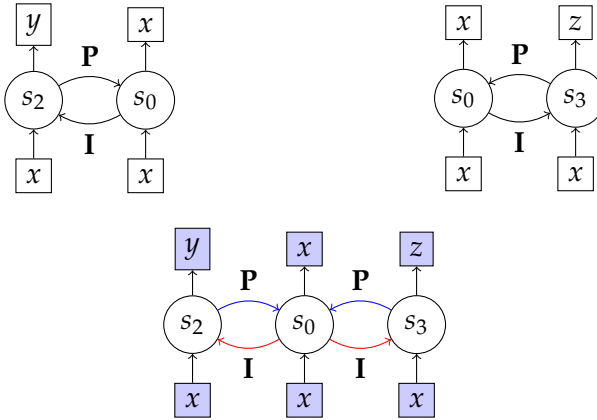


Figure 5.4: The two scope graphs seen at run-time for the Python example (above), and the combined scope graph with annotations (below). For both edges and vertices, a blueish color represents the D annotation and a red color the M annotation.

5.3.2 Context-Dependent Name Resolution

To solve the second problem, we introduce context-dependent name resolution, in which resolutions are performed in the context of an index. We use source locations as the index. The location captures the program location from where the resolution is performed, and affects the resolution results by only accepting declarations that have been declared before the program location from where the resolution occurs. In our example (Figure 5.3), this means that when we resolve from the program location `if ?`, we only find `x` to be in scope, since both `y` and `z` are defined after the `if ?` location. To achieve this, we extend scopes with a location component, and update the path rules from the resolution calculus to propagate a location that affects the available paths. The updated rules are displayed in Figure 5.5. Both the $ResD$ and $ResE$ rules now contain a comparison on indices, which results in the filtering of several paths that are present without this condition. The Vis rules also needs to be adapted, but only by including the location in the context.

$$\begin{array}{c}
\frac{k \vdash_G s \rightarrow x_i^D \quad i \leq k}{k \vdash_G \mathbf{D}(x_i^D) : s \mapsto (s, x_i^D)} \quad (\text{RESD}) \\
\frac{s'_{k_2} \notin S \quad k, S \vdash_G s_{k_1} \rightarrow_l s'_{k_2} \quad k_2 \leq k \quad k_1, \{s'_{k_2}\} \cup S \vdash_G p : s' \mapsto (s'', x_i^D) \quad \text{WF}(\mathbf{E}(l, s) \cdot p)}{k, S \vdash_G \mathbf{E}(l, s) \cdot p : s \mapsto (s'', x_i^D)} \quad (\text{RESE}) \\
\frac{\vdash_G x_i^R \rightarrow s \quad i, \{s\} \vdash_G p : s \mapsto (s', x_i^D)}{\vdash_G p : x_i^R \mapsto (s', x_i^D)} \quad (\text{RESR})
\end{array}$$

Figure 5.5: Resolution calculus extended with context-dependent name resolution via a visibility index.

5.3.3 Multi-Stage Resolution

More complex naming structures require multiple resolution steps. As shown earlier, an object access $x.y$ first requires the resolution of the variable x to determine its type and associated scope, which can then be used to resolve y . Determining the type in a dynamically typed language means to determine its value(s). Consequently, name resolution for dynamically typed languages depends on the values of variables. To be able to support more complex queries, we combine our previous extensions to the scope graph model with abstract interpretation.

5.4 OBTAINING SCOPE GRAPHS VIA HEAPS

To obtain an over-approximating scope graph while also having access to the types of values assigned to variables, we utilize abstract interpretation. In our abstract domain, we utilize the 1-to-1 correspondence between scopes and frames to obtain scope graphs from heaps. Our abstract interpretation results in an over-approximation of the abstract heaps seen at all program points. This provides us with a mapping from variables to values per program point. The abstract heaps are then translated into scope graphs and joined into one over-approximating scope graph.

5.4.1 Updated Heaps and Frames

The original definition of heap and frames uses the statically obtained scope graph for resolution. However, we do not have such a scope graph and need to modify the original definition. Our modification is given in Figure 5.7. A frame is thus a 4-tuple containing a scope id; a set of references made within the scope of the frame; a mapping from labels to frame ids, modeling connections between frames; and a mapping from declarations to values. A heap is a mapping from frame ids to frames. Two types of relations are defined: frame operations (\Rightarrow) and the resolution relation (\rightarrow). Frame operations operate in the context of a modifiable heap, which is represented by the $/h$ syntax. The resolution rules operate under an immutable heap (indicated by \vdash_h) and in the context of a set of seen frame ids (F).

The main differences with the original formulation are that the frame component is extended with a set of references (Σ) and that resolution of variables to declarations happens at run-time. This is reflected by the change in the *DLookup* rule, and the introduction of the *DPath* and *EPath* rules. The *DLookup* rule results in an *Addr* which points to a unique location in the heap via the frame id and the declaration, and contains the resolution path. The path can be used in the dynamic semantics of a language to decide whether an existing declaration needs to be overwritten or shadowed. The remaining two elements provide the required data for both the *get* and *set* operations to get and respectively set values at the specific location. The extended definition also includes the *Link* rule, which makes it possible to extend the dynamic links component of frames. Moreover, in contrast to the original definition, the available declarations are not fixed, as indicated by the removal of the requirement that a declaration is in the domain of $D_h(f)$ in the *set* rule.

Although we have removed the static scope graph dependency, we still have a *ScopeId* component in the definition of frames. This component is required to enable the translation from frames to scope graphs, since multiple frames can refer to the same scope, which is something we want reflected in the scope graph obtained from this translation: the information in all those frames needs to be captured by the shared scope in the resulting scope

Frames and Heaps

$$\begin{aligned}
f \in \text{FrameId} &= \{f_1, f_2, \dots\} \\
ks \in \text{DynLinks} &= \text{Label} \rightarrow \text{FrameId} \\
\sigma \in \text{Slots} &= \text{Decl} \rightarrow \text{Val} \\
\Sigma \in \text{Refs} &= \mathcal{P}(\text{Vars}) \\
\langle s, \Sigma, ks, \sigma \rangle \in \text{Frame} &= \text{ScopeId} \times \text{Refs} \times \text{DynLinks} \times \text{Slots} \\
h \in \text{Heap} &= \text{FrameId} \rightarrow \text{Frame}
\end{aligned}$$

Projection Functions

$$\begin{aligned}
\mathcal{S}_h(f) &= s & \mathcal{R}_h(f) &= \Sigma \\
\mathcal{K}_h(f) &= ks & \mathcal{D}_h(f) &= \sigma \\
&& \text{where } h(f) &= \langle s, \Sigma, ks, \sigma \rangle
\end{aligned}$$

Figure 5.6: Modified formal definition of the structural parts of frames and heaps [170].

graph. A language semantics thus needs to label frames with this information. In practice, the source location of frame producing constructs, such as functions, can be used as the scope id. With this in mind, we give a translation function (ϕ) that translate a heap into a scope graph, which uses a helper function (ψ) that translates a frame into a scope graph. $\text{Dom}_s(h)$ projects all frames with scope id s .

$$\begin{aligned}
\psi(\langle s, \Sigma, ks, \sigma \rangle) &= \{s \rightarrow (d, D) \mid \forall d \in \text{Dom}(\sigma)\} \\
&\cup \{(r, D) \rightarrow s \mid \forall r \in \Sigma\} \cup \{s \rightarrow_{(l, D)} s' \\
&\quad \mid \forall l \in \text{Label} \wedge \mathcal{S}(ks(l)) = s'\} \\
\phi(h) &= \bigcup \{ \bigsqcup \{ \psi(h(f)) \mid f \in \text{Dom}_s(h) \} \mid s \in \mathcal{S}(h) \}
\end{aligned}$$

The translation is sound with respect to definitely present edges, references, and declarations, which follows from the definition of the \leq_s relation. The translation is not sound on maybe present entities, i.e. the scope graph can contain resolution paths with M labels that are not present in the heap. Reasoning thus happens purely with the D annotated values.

Frame Modifications and Paths

$$\begin{array}{c}
 \frac{f' \notin \text{Dom}(h)}{\text{initFrame}(s, \Sigma, ks, \sigma) / h \Rightarrow f' / h [f' \mapsto \langle s, \Sigma, ks, \sigma \rangle]} \quad (\text{INITFRAME}) \\
 \\
 \frac{h' = K_h(f) [l \mapsto f']}{\text{link}(f, f', l) / h \Rightarrow h'} \quad (\text{LINK}) \quad \frac{x_j^D \in \text{Dom}(\mathcal{D}_h(f))}{\vdash_h \mathbf{D}(x_j^D) : f \rightarrow (f, x_j^D)} \quad (\text{DPATH}) \\
 \\
 \frac{x_j^D \notin \text{Dom}(\mathcal{D}_h(f)) \quad l \in \text{Dom}(\mathcal{K}_h(f)) \quad \mathcal{K}_h(f)(l) = f' \quad f' \notin F \quad F \cup \{f'\} \vdash_h p : f' \rightarrow (f'', x_j^D) \quad \text{WP}(p)}{F \vdash_h \mathbf{E}(l, f) \cdot p \rightarrow (f'', x_j^D)} \quad (\text{EPATH})
 \end{array}$$

Dynamic lookup

$$\frac{\{f\} \vdash_h p : f \rightarrow (f', x_j^D) \quad \exists k, p', f''. (p' : f \rightarrow (f'', x_k^D) \wedge p' < p)}{\text{lookup}(f, x_i^R) / h \Rightarrow \text{Addr}(f', x_j^D, p) / h [f \mapsto (\mathcal{R}_h(f) \mapsto \mathcal{R}_h(f) \cup \{x_i^R\})]} \quad (\text{DLOOKUP})$$

Slot value operations

$$\frac{x_i^D \in \text{Dom}(\mathcal{D}_h(f)) \quad \mathcal{D}_h(f)(x_i^D) = v}{\text{get}(f, x_i^D) / h \Rightarrow v} \quad (\text{GET})$$

$$\frac{}{\text{set}(f, x_i^D, v) / h \Rightarrow () / h [f \mapsto (\mathcal{D}_h(f) [x_i^D \mapsto v])]} \quad (\text{SET})$$

Figure 5.7: Modified formal definition of the behavior of frames and heaps [170].

$x, fn \in Var$	(variables, function variables)
$p \in P ::= l$	(programs)
$l \in Sl ::= s \mid l \mid \epsilon$	(lists of statements)
$s \in S ::= x = e \mid \text{if}(e) \{l_1\} \text{ else } \{l_2\} \mid \text{def } fn(x) \{ l \}$	(statements)
$e \in E ::= e_1 - e_2 \mid 1 \mid x \mid fn(e)$	(expressions)

Figure 5.8: Grammar of a simple procedural language.

5.4.2 Natural Semantics with Heap and Frames

The modifications to the heap and frames definition requires additional bookkeeping in the dynamic semantics of a language. Frames need to be annotated with their scope id, and modified heaps need to be propagated. Also, the dynamic semantics needs to decide when a declaration is constructed versus when an existing declaration is used. To illustrate these adjuncts, we introduce a small dynamically typed language à la Cousot [47] with functions, which we utilize as a running example throughout the rest of this paper. The concrete syntax specification is given in Figure 5.8, and the dynamic semantics is given in Figure 5.9. As in prior work [170], we define the dynamic semantics in the context of a current frame id and the current heap: $f \vdash e/h \Rightarrow v/h'$ means that program e evaluates in the context of heap h with the current frame identified by f to v and updated heap h' .

Most of the resolution is hidden within the frames definition. In the dynamic semantics we only differentiate between declarations and modifications. In our example language, this is done with the *S-assign* and *S-assign-new* rules. The *S-assign-new* rule creates a declaration in the current frame, while the *S-assign* rule updates an existing declaration. In addition, most rules of the dynamic semantics now modify the heap, as indicated by the change in subscripts. This is especially apparent in the *E-min* rule. Expressions generally do not alter the heap. But in our case, heap modifications might arise via expressions due to variable references, which extend the Σ set of the current frame. The *Def-*

$$\begin{array}{c}
f \vdash e/h_1 \Rightarrow v_1/h_2 \quad \text{lookup}(h_2, f, x) \not\Rightarrow \\
\hline
\text{set}(h_2, f, x) \Rightarrow h_3 \quad \text{(S-ASSIGN-NEW)} \\
f \vdash x = e/h_1 \Rightarrow h_3 \\
f \vdash e/h_1 \Rightarrow v_1/h_2 \quad \text{lookup}(h_2, f, x) \Rightarrow \text{Addr}(f', x_i^D, p)/h_3 \\
\hline
\text{set}(h_3, f', x) \Rightarrow ()/h_4 \\
\hline
f \vdash x = e/h_1 \Rightarrow h_4 \quad \text{(S-ASSIGN)} \\
f \vdash E_1/h_1 \Rightarrow V_1/h_2 \quad f \vdash E_2/h_2 \Rightarrow V_2/h_3 \\
\hline
V_1, V_2 \in \mathbb{Z} \quad \text{(E-MIN)} \\
f \vdash E_1 - E_2/h_1 \Rightarrow V_1 - V_2/h_3 \\
\text{set}(h_1, f, \text{fn}, \text{Clos}(x, sl, l, \text{fn})) \Rightarrow ()/h_2 \quad \text{(DEF-FN)} \\
\hline
f \vdash (\text{def fn}(x)\{sl\})_1/h_1 \Rightarrow h_2 \\
f \vdash e_1/h_1 \Rightarrow v_1/h_2 \quad \text{lookup}(h_2, f, \text{fn}) \Rightarrow \text{Addr}(f', \text{fn}^D, p)/h_3 \\
\text{get}(h_3, f', \text{fn}^D) \Rightarrow \text{Clos}(x, b, l, \text{fn}_c) \\
\text{initFrame}(l, \{\}, \{(P, \text{fn}_c)\}, \{(x, v_1)\})/h_3 \Rightarrow f''/h_4 \\
\hline
f'' \vdash b/h_4 \Rightarrow v_2/h_5 \\
\hline
f \vdash \text{fn}(e_1)/h_1 \Rightarrow v_2/h_5 \quad \text{(E-CALL)}
\end{array}$$

Figure 5.9: Big-step operational semantics of part of our running example with explicit frame and heaps.

f_n rule demonstrates the choice of scope ids when constructing frames. In the rule, the source label, indicated by the l subscript, is stored in the closure (identified by $Clos$). When calling a function, described by the $E-call$ rule, the source label is used as the scope id for the newly created frame in which the body of the function is evaluated. Multiple calls to the same function thus result in different frames that share the same scope id.

5.5 ABSTRACT INTERPRETATION WITH HEAP AND FRAMES

Having defined heap and frames without an underlying scope graph, we focus on abstract interpretation using this new definition. We provide a language-independent abstraction, which is parameterized by an abstraction for values.

5.5.1 *Safe Heap and Frames*

We provide a language-independent abstract definition of heap and frames, and define a safety relation between concrete and abstract heap and frames. Using this safety relation, we obtain a Galois connection between the concrete and abstract definitions. In our abstract definition, we restrict the number of frame ids to a finite amount. This requires removal of the $f' \notin Dom(h)$ premise in the abstract definition of the $InitFrame$ rule. Moreover, the slots function becomes a mapping from declarations to abstract values, with the requirement that the abstract values domain forms a complete lattice. We also annotate declarations, references, and labeled edges in our abstract frames. The annotations necessitate a modification to the ψ and ϕ functions to utilize the annotations of the frames instead of always assigning the D annotation to scopes, declarations, and references. From now on, we are working with those versions when referencing the ϕ or ψ function. Besides annotating frame components, we also define an ordering on abstract frames that is almost identical to the ordering defined on annotated scope graphs, but altered to align with the heap and frames definition, and extended with a constraint on slots: $\forall d \in Dom(\sigma_1). \sigma_1(d) \leq \sigma_2(d)$, where σ_1 and σ_2 are the slots functions for the two frames under comparison. Furthermore, we require that the associated scopes of the two

compared frames are equal, and the ordering is extended to a lattice. Finally, with our abstract definition, we define a safety relation between concrete heaps and abstract heaps.

$$h_c \text{ safeHeap } h_a \text{ iff} \\ \forall f \in \text{Dom}(h_c). \exists f_a \in \text{Dom}(h_a). h_c(f) \text{ safeFrame } h_a(f_a),$$

$$\langle s, \Sigma, ks, \sigma \rangle_{h_c} \text{ safeFrame } \langle s_a, \Sigma_a, ks_a, \sigma_a \rangle_{h_a} \text{ iff} \\ s = s_a \wedge \Sigma \subseteq \Sigma_a \wedge D_\Sigma(\Sigma_a) \subseteq \text{Dom}(\Sigma) \\ \forall l \in \text{Dom}(ks). h_c(ks(l)) \text{ safeFrame } h_a(ks_a(l)) \\ \forall d \in \text{Dom}(\sigma). \sigma(d) \text{ safeVal } \sigma_a(d) \\ D_\sigma(\sigma_a) \subseteq \text{Dom}(\sigma) \wedge D_{ks}(ks_a) \subseteq \text{Dom}(ks)$$

The *safeFrame* relation is parametric in the language-specific *safeVal* relation, and recursive, so the largest set satisfying the relation is required. D_Σ projects the definitely present references, D_σ projects the definitely present declarations, and D_{ks} projects the definitely present links.

We define the collection semantics as a function

$$\text{coll}_t p : \text{ProgramPoint} \rightarrow \mathcal{P}(\text{Heap} \times \text{Heap})$$

from program points to heaps, with t the tree representing an execution.

$$\text{coll}_t(p) = \{(h_1, h_2) \mid f \vdash p/h_1 \Rightarrow v/h_2 \text{ is a state in } t\} \\ \cup \{(h_1, h_2) \mid f \vdash p/h_1 \Rightarrow h_2 \text{ is a state in } t\}$$

The collection semantics collects all heap pairs seen at a program point, with which we can obtain the pair of over-approximating scope graphs for a program point:

$$\mathcal{G}_t(p) = (\bigsqcup \{\phi(h_1) \mid (h_1, h_2) \in \text{coll}_t p\}, \bigsqcup \{\phi(h_2) \mid (h_1, h_2) \in \text{coll}_t p\}).$$

5.5.2 Reasoning with Over-Approximated Scope Graphs

Using the over-approximated scope graphs defined previously, we can define auto-completion services. When the evaluation is productive — $(h_1, p) \Rightarrow h_2 \implies G(h_1) \leq G(h_2)$ — we can take the resulting heaps of

the root of the computation tree and reason with the obtained scope graph pair.⁴ Otherwise, reasoning does not happen on one scope graph, instead it happens on the scope graph of the program point from where we are reasoning. The advantage of reasoning with one scope graph, is that edit operations can be interpreted as a modification on the single scope graph. As a result, the editor service does not need to re-evaluate the program on every edit. Auto-completion candidates at program point p for computation tree t are given by the visible declarations that are definitely present, where $S(p)$ gives the scope associated with program point p .

$$AC_t(p) = \{x_i^D \mid \vdash_G S(p) \rightarrow (s', (x_i^D, D))\}$$

5.6 IMPLEMENTATION

We have implemented our approach in Haskell, providing a library for the construction of concrete and abstract interpreters with explicit heap and frames operations.

5.6.1 Heaps and Frames

The main component of our implementation is the encoding of our formalized heap and frames model. A substantial part of this encoding is a simple 1-to-1 translation from the formal model to Haskell code. Heap and frames are implemented as parameterized data types, with the parameters representing the type for frame ids, scope ids, type of declarations, and the type of values.

```
data Heap f s d v = Heap (Map f (Frame f s d v))
data Frame f s d v = Frame
{sid :: s, refs :: Set d, ks :: Map Label f, slots :: Map d v}
```

The operations on heap and frames follow the formal model and are defined in terms of the parameterized data types. One difference is the handling of frame ids. In our implementation, fresh frame ids need to be provided by the language, since the implementation is parametric in the type of frame ids.

5.6.2 Abstract and Concrete Interpreters

The abstract and concrete interpreters are language-specific. To ease the construction of these interpreters, we provide several helper functions around a systematic approach based on Sturdy [97], which splits

⁴ The used ordering for productivity is left abstract.

a language implementation into three parts: a generic interpreter, a concrete interpreter, and an abstract interpreter. The generic interpreter describes functionality that is common among all interpretations, and is defined in terms of indeterminate operations. These operations are determined by the concrete and abstract interpreters, resulting in a working interpreter in the respective domain. We define an abstract signature for the operations of the language using type classes. Instances of the type class then give an interpretation to the signature.

5.6.3 *Collecting Semantics and Termination*

We utilize lenses [72] to get access to the parameterized heap from the opaque interpreter context. Interpreters are written in an open recursive style, where recursive evaluations are handled by continuation functions. We provide a generic continuation function that annotates program points with the current heap and ensures termination via the productive caching algorithm [57]. Program points are identified by a label, which we obtain via a type class. To sequence computations we utilize monads [140, 214]. Our continuation function is thus parametric in the monad it evaluates in. The final set of annotations corresponds to the collecting semantics.

5.7 EXPERIMENT DESIGN AND RESULTS

To demonstrate our approach, we implement a small subset of Python and compare the auto-completion candidates given by our approach with the state of the art auto-completion service providers for Python. To ensure correctness of our implementation, we utilize the existing Python (3.12.3) implementation as an oracle. We have chosen Python because it is a popular programming language⁵ and has several editor service implementations, making it a suitable vehicle for us to provide a compelling example of our approach. Our subset supports functions with parameters, classes and objects, primitive types and operations on those types, if-else statements, and while loops.

We have constructed a test set of programs around our subset with key name binding challenges for completion services [73]. In our experiment, we filter the possible completion candidates by prefixing all variables with an x , and only completing on variables starting with an x . This is to prevent the completion list from being filled with standard Python constructs, such as `__name__`.

⁵ <https://survey.stackoverflow.co/2024/technology#2-programming-scripting-and-markup-languages>

To implement an abstract interpreter for our Python subset, we use the approach from prior work [141], which performs type checking using abstract interpretation. Our abstract domain thus maps values to their type. The modeling of the heap is slightly different, since we use our heap and frames approach, which is a less compact representation.

To determine in-scope variables, we execute the Python interpreter and execute the `dir` and `globals` functions, which provide access to the available names in the current scope and the global variables currently in scope, respectively. The `dir` function can also be used to obtain the attributes of an object. We execute our programs with these functions placed at the location where we will perform an auto-completion request to obtain the variables in-scope at run-time at the specific location. We run the programs under a variety of inputs to ensure 100% path coverage and then take the intersection of the variables obtained from these executions, which corresponds to the set of variables that are present among all possible paths. In case of the `fn-not-called` program, we obtained in-scope variables by utilizing the respective Python file as a library and testing the function externally.

We have chosen three auto-completion service providers for Python: Pylance, PyCharm⁶ (professional edition), and Jedi. Pylance is an implementation of the LSP for Python and uses Pyright⁷, a static type checker for Python; PyCharm is a Python IDE developed by JetBrains, which provides Python support via a plugin⁸; and Jedi is a static analysis tool for Python with a focus on auto-completion and goto-definition.

As not all implementations have a public API, we have performed our experiments manually, via VSCode⁹ for Jedi and Pylance, and using the PyCharm GUI for PyCharm. For our approach, we have implemented the auto-complete part of the LSP protocol and also used interactions via VSCode to evaluate our approach.

5.7.1 Results

The results of our experiment are displayed in Table 5.1, as precision and recall pairs. A precision of 100% equates to soundness. A recall of 100% equates to completeness. In the table, P describes the number of sound completion candidates, and $P + N$ describes the number of sound and unsound completion candidates. This number is constrained by the variables in a program. Modifying the values of P and N can result in a different percentages for precision and recall, but the relation with respect to P and N stays consistent.

⁶ <https://www.jetbrains.com/pycharm/>

⁷ <https://github.com/microsoft/pyright>

⁸ <https://plugins.jetbrains.com/plugin/631-python>

⁹ <https://code.visualstudio.com/>

The results highlight a strategy present in the current state of the art: completeness is preferred over soundness. Consequently, even on simple programs, the state of the art introduces completion candidates that can introduce name binding errors when selected by the user. Our approach prefers soundness over completeness, and presents no unsound completion candidates with respect to name binding for programs in our experiment. Additionally, among the state of the art we observe similar results. The small differences can be explained by the underlying approach of the different services, or by the development hours put into a service. Another observation is that recall is generally around a 100%. In most cases, obtaining a high percentage for recall can be easy, because an editor service can collect all variables in the program and present those as completion candidates.

Some of the programs in our demonstration could also be handled by the state of the art tools by strengthening their analyses without adding significant complexity. There are also programs, such as *add-field*, which require more complex analysis and the gain of soundness on such candidates might not be worth the increase in complexity. Nevertheless, the set of evaluated programs gives an overview of key name binding constructs on which analysis can be improved in the state of the art auto-complete services for Python.

5.8 DISCUSSION

In this section we compare our approach to the state of the art, and give several suggestions for future work, including steps towards (full) soundness.

5.8.1 *Missing Candidates and Uncalled Functions*

Our approach misses several valid completion candidates. For example, the *if-else1* program contains a condition that is always false, which is detected by Jedi. Since we used an abstract domain that maps values to types, we lose such information, which can lead to lower recall. This is also shown by the program *if-else-both*, where both branches of an *if-else* are taken by repeating the statement but taking a different branch. Due to our chosen abstract domain, our approach fails to determine that the variables introduced in both branches are definitely in scope. This can be solved by modifying the abstract domain. However, as is inherent to abstract interpretation, there is a trade-off between performance and the precision of the abstract domain. In future work, we would like to investigate the effects of different abstract domains on the recall and performance.

Table 5.1: Results of our experiment on the state of the art on the Python test set. P denotes the sound completion candidates. $T = P + N$ denotes the total available completion candidates, so sound and unsound. Precision (Pr) describes the percentage of completion candidates that were sound compared to all given completion candidates by a tool. Recall (Re) describes the percentage of sound completion candidates that were reported by the tool compared to all possible sound completion candidates at the program point. N/A indicates that a tool gave no completion candidates.

Program	$P : T$	PyCharm		Jedi		Pylance		Our work	
		Pr	Re	Pr	Re	Pr	Re	Pr	Re
<i>self-ref</i>	1 : 2	50%	100%	50%	100%	50%	100%	100%	100%
<i>if-else1</i>	2 : 3	66.7%	100%	100%	100%	66.7%	100%	100%	50%
<i>if-else2</i>	1 : 3	33.3%	100%	33.3%	100%	33.3%	100%	100%	100%
<i>if-else3</i>	2 : 3	100%	100%	66.7%	100%	66.7%	100%	100%	100%
<i>if-else-both</i>	3 : 3	100%	100%	100%	100%	100%	100%	100%	33%
<i>duck-type</i>	1 : 3	33.3%	100%	33.3%	100%	33.3%	100%	100%	100%
<i>use-before-define</i>	1 : 2	100%	100%	100%	100%	50%	100%	100%	100%
<i>use-before-define-fn</i>	1 : 2	50%	100%	50%	100%	50%	100%	100%	100%
<i>add-field</i>	2 : 2	100%	50%	100%	50%	100%	50%	100%	100%
<i>fn-not-called</i>	2 : 2	100%	100%	100%	100%	100%	100%	N/A	N/A
<i>obj-param</i>	1 : 3	33.3%	100%	33.3%	100%	N/A	N/A	100%	100%
<i>global-add</i>	2 : 2	100%	50%	100%	100%	100%	100%	100%	100%
<i>not-global-add</i>	1 : 2	100%	100%	50%	100%	50%	100%	100%	100%

A major drawback of our approach is that uncalled functions have an associated empty scope. Ergo, completion candidates within such a function are limited to what is available via the parent scope. This is tested by the *fn-not-called* program. The state of the art has no problem with this specific case. In future work we aim to investigate suitable conditions under which we can safely simulate the calling of these functions to still obtain an over-approximated scope graph.

5.8.2 *Being Fully Sound*

In this work we have made a first step towards our goal, which is to obtain sound editor services. To attain it, we need to fully formalize our approach and prove that our heap and frames operations respect the safety relations. We are currently in the process of proving our approach using Agda [151]. Furthermore, languages need to prove that their abstract evaluation relation is sound by showing that it respects that safety relation.

During informal discussions with Python developers, they indicated that unsound completion candidates can be helpful during exploration or refactoring. Hence, just providing sound completion candidates might hamper developers during such activities. We overcome this by retaining the possibly unsound completion candidates. These can still be provided as candidates by utilizing the annotations to provide developers with more information. For example, by giving unsound suggestions an indicator to communicate that they might introduce erroneous execution paths. This would also provide a fallback option for uncalled functions. In future work, we aim to investigate this further and compare the user experience via user studies.

5.8.3 *Incremental Analysis and Imports*

The original scope graph framework supports module imports. We have omitted this to simplify the presentation. However, we do not see any theoretical difficulties in supporting module imports with an extra rule to the resolution calculus. However, difficulties with modules might arise with respect to scalability and usefulness. Building detailed scoping information for modules which are not modified by the programmer is not beneficial. Nevertheless, imported functions can modify variables in such a way that they affect the scoping in the module in which the programmer is working. Think of a function that adds fields to an object. To handle this, we want to investigate capturing the transformations made by imported functions using predicate transformer semantics [60].

5.8.4 *Handling of Incorrect Programs*

In this paper we have worked under the assumption that programs are correct, while editor services often operate on broken programs or programs with holes. In case of broken programs, we can continue our abstract interpretation but mark all results as unsound, and show them with an annotation to indicate the possible introduction of erroneous execution paths. For programs with holes, a language can define the semantics and do the interpretation over such programs. However, it is unclear how scalable this is. Alternatively, language designers can define a projection from programs with holes to programs without . The interpretation is then performed on the program without holes and the results can be applied on the program with holes. Furthermore, for simple editing operations, we can modify the scope graph without re-interpretation of the program, and can therefore work with holed programs. We thus do not see any substantial difficulties in adapting our approach to handle incorrect program and programs with holes.

5.8.5 *Alternative Editor Services*

In this paper we have focused on auto-completion services, because completeness is less important for such a service. Missing a completion candidate will not break a program. For other services, such as refactoring, not being complete is a problem. Missing a case during a refactoring can break a program, or worse, change its behavior. Nevertheless, our model provides enough information to implement alternative services. For example, the find (all) references service. However, we might not report all references, due to incompleteness. In future work we plan to investigate if there are scenarios in which we can be complete.

5.9 RELATED WORK

Scope graphs [149] have been used as a blueprint for dynamic memory [170]. Van Antwerpen et al. [5, 6] have used scope graphs to perform static analysis of types, initialization, and name binding in a language-independent manner. Zwaan et al. [224] give a detailed overview of these developments.

Scope graphs have also been used to obtain language-parametric editor services for statically typed languages [159]; to preserve well-typedness during automated refactoring [137]; and in the construction of completion services for statically typed languages [158] that support both syntactic and semantics completions. Compared to our work, we do not see an immediate way to obtain preservation of well-typedness

for automated refactorings, due to incompleteness. With regards to completion services, prior work focused on statically typed languages and used grammars and type specifications. Our work requires an abstract interpreter, and is purely focused on semantic completions and dynamically typed languages.

Stack graphs [49] is a modification of scope graphs that support file-incremental analysis with type-dependent look ups, and powers code navigation at GitHub. The approach is mostly focused on code navigation, which boils down to resolving references to declarations. To be able to support new languages with ease, the authors have constructed a graph construction language on top of the tree-sitter parser framework. Languages which have a tree-sitter parser can define patterns that map language constructs to operations on stack graphs, independent of the type of language. To handle more complex situations, the approach uses data-flow analysis. However, it is unclear whether the data-flow semantics is extracted out of patterns or requires additional effort. With our approach, the primary focus is on completion services, and no additional effort is required to support more complex name resolution scenarios. However, the initial effort required by our approach is much more substantial due to the need of a working abstract interpreter.

5.9.1 *Data-Flow Analysis and Abstract Interpretation*

Many editor services use some kind of data-flow analysis [100, 150] to support more complex scenarios. For example, use-definition chains [98] can be used to determine to which declaration(s) a reference belongs. The monotone framework [93] provides a reusable pattern for defining data-flow analysis in a systematic manner.

Data-flow analysis and abstract interpretation are tightly connected [189, 191]. Our approach could be more data-flow oriented, using the monotone framework. Nevertheless, the usage of abstract interpretation in combination with the code structuring technique promoted by Sturdy [97], provides much opportunity for reuse between the interpreters.

Prior work combined abstract interpretation with statistical models to provide completion candidates [171]. The approach was evaluated on Java, and only a small percentage of the completion proposals gave type errors. It is unclear whether that included name binding errors, and how it performs on a dynamically typed language.

5.9.2 *Code Completion Using Machine Learning*

In recent years, code tasks, such as auto-completion, have received significant attention from the machine learning community [220]. Transformers [211] have been used by Kim et al. [102] to improve candidates over pre-existing machine-learning based techniques, and shows promising results, but incorrect predictions are possible. Besides just giving completion suggestions, modern systems are capable of more by synthesizing snippets from comments or from context, such as GitHub Copilot¹⁰ and Tabnine¹¹. We have not included such systems in our comparison since the tasks performed by these systems is rather different and requires a different form of evaluation [223].

5.10 CONCLUSION

In this chapter we have investigated the construction of auto-completion services for dynamically typed languages, such that the given completion candidates do not cause name binding errors. To achieve this, we have extended the scope graph framework with annotations and context-dependent name resolution. Furthermore, we have used the 1-to-1 correspondence between scopes and frames in combination with abstract interpretation to obtain a sound over-approximation of the name binding seen at run-time. To demonstrate our approach, we applied it to a small subset of Python and compared it to the state of the art editor services. On this test set, our approach outperformed the state of the art with respect to the sound completion candidates, and sometimes also with respect to completeness. However, uncalled functions present a difficulty for our approach, missing locally valid completion candidates. Finally, we have discussed the steps needed to obtain auto-completion services that are fully sound with respect to name binding.

¹⁰ <https://github.com/features/copilot/>

¹¹ <https://www.tabnine.com/>

In this chapter we look at another type of tooling used by developers: debuggers. Our focus will be on debuggers for non-deterministic languages. By focusing on non-deterministic languages, we automatically obtain debuggers for deterministic languages, and debugging non-deterministic languages is somewhat reminiscent to exploratory programming, which we explored in Chapter 2. Additionally, investigating debuggers for non-deterministic languages is interesting due to the inherent difficulty of this task, primarily caused by the state space explosion problem, making such debugging tasks infeasible to explore manually and automatically. Multiverse debugging addresses these problems by realizing a fine-grained, exhaustive and interactive process for state space exploration. Essentially, a user-guided search through the state space of a particular program, with options to automate parts of this search. In this chapter we collect requirements for multiverse debuggers through three case studies, one of which is funcons. Based on these requirements, we define a more general framework for multiverse debugging compared to the state of the art. Having funcons as one of our case studies, the resulting debugger is applicable to languages defined with *iCoLa*⁺.

Associated Publications

- **Damian Frölich**, Tommaso Pacciani, and L. Thomas van Binsbergen. “Exploratory, Omniscient, and Multiverse Diagnostics in Debuggers for Non-Deterministic Languages.” In: *Proceedings of the 18th ACM SIGPLAN International Conference on Software Language Engineering*. SLE '25. Koblenz, Germany: Association for Computing Machinery, 2025, pp. 134–147. ISBN: 9798400718847. DOI: 10.1145/3732771.3742719

6.1 INTRODUCTION

Conventional **stepwise debuggers** can be used to explore the execution of a program (a run) in a step-by-step manner, giving programmers control to interrupt and proceed execution as they see fit, and enabling them to inspect concrete information about that run at the moment of interruption (e.g., active bindings, variable assignments, object state). The level of granularity of the steps, the control mechanisms, and the

observable context information depends per language, but typically involves setting breakpoints on program locations and monitoring mutations to specific variables.

Omniscient debuggers (also referred to as ‘back-in-time debuggers’) extend stepwise debuggers by recording information during a debug session to allow programmers to revisit some or all steps of the execution [33, 121]. This functionality is particularly useful to understand how a particular (undesirable) program state came to be by retracing steps and the evolution of variables and (other) objects (without having to anticipate meaningful intermediate states beforehand by setting up breakpoints and monitors).

Non-deterministic programs admit multiple runs, each of which can (potentially) exhibit different desirable or undesirable behavior (bugs). Debugging non-deterministic programs is inherently challenging as the set of possible runs may be vast, hard to predict, and may contain runs that occur only rarely. Conventional debuggers provide only a partial view on the bugs a program admits as they perform a single run per debugging session. Repeated debugging sessions are required and rare bugs may remain unobserved.

To address these challenges, **multiverse debuggers** provide an interactive, user-controlled and simultaneous exploration of multiple runs, avoiding redundant work by detecting syntactically equal states [125]. Pasquier et al. [156] introduced user-defined reductions over states, giving the user a mechanism to reduce the explored state space. The authors define a language-parametric framework for obtaining omniscient, multiverse debuggers through the definition of a transition relation for the object language. In [157], the authors introduce a generic breakpoint-specification mechanism and demonstrate various logics (such as regular expressions and linear temporal logic) for expressing traces of interest and further inspection.

Exploratory programming [99, 173, 206] has been mentioned several times already in this thesis. To recap: it is a programming style where the goal worked towards is open ended. Exploratory programming generalizes incremental programming in way similar to how multiverse debugging generalizes stepwise debugging: multiple runs are explored interactively and in parallel in order to investigate (un)desirable outcomes. In exploratory programming, a programmer may additionally be interested in directly comparing the effects of multiple runs. Additionally, the definition of *runs* differs, where exploratory programming looks at sequences of programs as a run, and multiverse debugging looks at the intermediate steps of the execution of a program. Thus multiverse debugging operates at a finer level of granularity compared to exploratory programming.

In this work, we introduce the execution graph to an extended version of the framework of Pasquier et al. and generalize the concepts of breakpoint and reduction to admit additional user scenarios inspired by exploratory programming. We evaluate the applicability of multiverse debugging and the generality of the original and extended framework by using three case studies: grammar exploration, formal operational semantics, and reasoning with norms. As part of these case studies, we collect requirements for exploratory, omniscient, multiverse debuggers, forming the basis of our evaluation. Concretely, we make the following contributions.

- We (re-)define the original framework by Pasquier et al. using set-theoretic notation (in §6.2).
- We investigate the application of exploratory, omniscient, multiverse debugging in three domain-specific languages, resulting in a set of user stories and requirements for each of the domains (in §§6.3-6.5).
- We evaluate the applicability of the framework against the requirements, providing further evidence in support of the use of a generic framework, whilst identifying a number of limitations (in §6.7).
- We introduce an extended version of the framework that (partially) addresses these limitations (in §6.6).

§§6.9 - 6.11 discuss the threats to validity of our work, related work and conclude (respectively).

6.2 THE ORIGINAL MULTIVERSE FRAMEWORK

Our work builds on the reusable multiverse debugging framework introduced by Pasquier et al. [156]. We define their framework using set-theoretic notation as an alternative to the Lean [148] code given in the original paper, to ensure consistency with the notation used for the extensions we propose.

An overview of the debugging framework and the relations between different roles is displayed in Figure 6.1.

Definition 6.2.1. A Semantic Transition Relation (STR) is a tuple $\langle C, C_0, A, I, Act \rangle$, where C is a set of configurations, $C_0 \subseteq C$ a set of initial configurations, A is a set of actions, $I : C \times A \rightarrow \mathcal{P}(C)$ is a non-deterministic interpreter of actions upon configurations, and $Act : C \rightarrow \mathcal{P}(A)$ determines the set of executable actions for a given configuration.

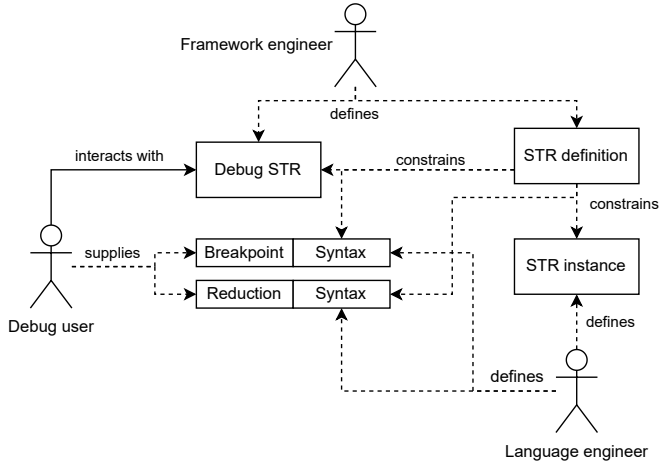


Figure 6.1: The relations between roles and components of the multiverse debugging framework.

There are two contributors to non-determinism in an STR: (1) for every configuration, there might be multiple executable actions (Act) and (2) for every action acting upon a configuration, there might be multiple result configurations (I).

Let $S = \langle C_s, C_{s0}, A_s, I_d, Act_s \rangle$ be a language STR, a multiverse debugger is defined in terms of this STR as follows:

$$D_S(R, B) = \langle C_d, C_{d0}, A_d, I_d, Act_d \rangle.$$

Parameter $R : C_s \rightarrow C_s/r$ is a reducer function reducing configurations to a reduced form C_s/r which support equality between elements. Parameter $B : C_s \rightarrow \mathbb{B}$ represents a breakpoint as a predicate over configurations, determining the configurations in which the breakpoint is 'activated'.

The configuration of a debugger is defined as a tuple:

$$C_d = \langle C_s \cup \{\perp\}, \mathcal{P}(C_s), \mathcal{P}(C_s) \rangle.$$

The first component denotes the current, object language-specific configuration or is \perp when there is none. The second component is a history represented as a set of (previously) encountered configurations. The last component is a set of options to choose from after the (non-deterministic) execution of an action.

$$\begin{array}{c}
\frac{a_s \in \text{Act}_s(c_s) \quad c_s \neq \perp \quad \text{opts} = I_s(c_s, a_s)}{\langle c_s, \text{hist}, _ \rangle \xrightarrow{\text{step } a_s} \langle \perp, \text{hist}, \text{opts} \rangle} \quad (\text{STEP}) \\
\\
\frac{c_s \in \text{opts}}{\langle _ , \text{hist}, \text{opts} \rangle \xrightarrow{\text{select } c_s} \langle c_s, \{c_s\} \cup \text{hist}, \emptyset \rangle} \quad (\text{SELECT}) \\
\\
\frac{c_s \in \text{hist}}{\langle _ , \text{hist}, _ \rangle \xrightarrow{\text{jump } c_s} \langle c_s, \{c_s\} \cup \text{hist}, \emptyset \rangle} \quad (\text{JUMP}) \\
\\
\frac{c_s \neq \perp \quad \text{find}_{(R,B)}(\{c_s\}) = (c_{s1}, \dots, c_{sn})}{\langle c_s, \text{hist}, _ \rangle \xrightarrow{\text{run_to_breakpoint}} \langle c_{s1}, \text{hist} \cup \{c_{s1}, \dots, c_{sn}\}, \emptyset \rangle} \quad (\text{RUN}) \\
\\
\frac{\text{find}_{(R,B)}(\text{opts}) = (c_{s1}, \dots, c_{sn})}{\langle \perp, \text{hist}, \text{opts} \rangle \xrightarrow{\text{run_to_breakpoint}} \langle c_{s1}, \text{hist} \cup \{c_{s1}, \dots, c_{sn}\}, \emptyset \rangle} \quad (\text{RUN-2})
\end{array}$$

Figure 6.2: Semantics of the debugging operations. Under scores denote unused meta-variables, and can be replaced by an appropriate meta-variable as long as every underscore gets assigned a unique meta-variable. The subscript s denotes components of the underlying language STR.

The debugger actions, set of available actions, and the interpreter are defined as follows. The interpreter is defined in terms of a transition relation \xrightarrow{a} , defined by the inference system in Figure 6.2.

$$\begin{aligned}
A_d &::= \text{step } A_s \mid \text{select } C_s \mid \text{jump } C_s \mid \text{run_to_breakpoint} \\
\text{Act}_d(\langle c_s, h, o \rangle) &= \{ \text{step } a_s \mid a_s \in \text{Act}_s(c_s), c_s \neq \perp \} \\
&\quad \cup \{ \text{jump } c \mid c \in h \} \cup \{ \text{select } c \mid c \in o \} \\
&\quad \cup \{ \text{run_to_breakpoint} \} \\
I_d(c, a) &= \{ c' \mid c \xrightarrow{a} c' \}
\end{aligned}$$

For any (stepwise) interpreter I , we define the reachability graph as embedding all the possible execution traces from a given configuration. The definition is adapted from [31].

Definition 6.2.2. Let I_a be an interpreter for actions $a \in A$ and configurations $c \in C$. The *reachability graph* from a configuration $c \in C$ is the graph $\langle V, E \rangle$ with V and E the smallest sets of nodes and labeled edges such that $c \in V$ and for every triple $\langle c_1, a, c_2 \rangle$, with $c_1 \in V$ and $c_2 = I_a(c_1)$, it holds that $c_2 \in V$ and that $\langle c_1, a, c_2 \rangle \in E$.

The semantics of *run_to_breakpoint* is defined in terms of *find* (not defined here) performing a depth-first search in the reachability graph

to find a configuration for which the B predicate succeeds. Throughout this search, a set of reduced configurations is maintained, containing the reduced versions of the configurations encountered during search by applying reduction function R . When a reduced configuration is revisited, the current search-branch terminates and *find* backtracks. If a configuration satisfying B is found, this configuration and its predecessors are returned as a sequence. If the search is exhausted, the empty sequence is returned. The reductions can yield a finite exploration of an infinite reachability graph. However, the algorithm does not terminate in the general case. See [156] for a more formal definition of *find*.

The functions R and B are the result of (partially) evaluating a breakpoint and reduction expression given by the programmer. The syntax for breakpoint- and reduction-expressions is determined by the language engineer (see Figure 6.1). The implementation of *find* in the original framework [156], described above, focused on breakpoints as predicates over configurations. The implementation needs to be modified for more expressive breakpoints. The implementation of *find* in [157] adds support for breakpoints over transitions and sequences of transitions with regular expressions and LTL-formulae as example formalisms.

6.3 GRAMMAR AND PARSER ENGINEERING

In this and the following two sections we present case studies across three different domains: grammar engineering, formal operational semantics, and norm engineering. For every domain we investigate usage scenarios in which a user attempts to locate, understand, and consider resolutions for (together: *diagnose*) a particular error, bug, or otherwise unwanted result. To this end, we describe each domain, define the most important user roles, and associate one or more user stories with each of the roles. The user stories are re-formulated as functional requirements for debuggers and, by extension, for an underlying debugging framework. The requirements are derived from the needs of (hypothetical) users of domain-specific debugger implementations and are used in §6.7 to evaluate the multiverse debugging frameworks discussed in this paper. The first case study investigates diagnosis in the context of grammar and parser engineering.

A context-free grammar (simply ‘grammar’, hereafter) specifies the concrete syntax of a (software) language. The conventional definition of a grammar, originally provided by Chomsky [40], associates one or more production rules with nonterminal symbols. A production rule consists of a sequence of nonterminal symbols and terminal symbols, with terminal symbols capturing the tokens (words) of a language.

A nonterminal in a grammar derives sentences (sequences of tokens) through the recursive process of in-place replacing nonterminal symbols – with one of the productions associated with that nonterminal – until a sequence consisting of only terminal symbols is obtained.

This process is non-deterministic when at least one of the encountered nonterminal symbols has two or more associated production rules. As a result, the same nonterminal can be used to generate multiple sentences. Conversely, the same sentence can be the result of alternative sequences of derivation steps starting from the same nonterminal. Grammars that have one or more such sentences¹ are *ambiguous*. The possibly many combinations of non-deterministic choices in the derivation process is the source of the great expressiveness of grammars, but also of (any) complexity in parsers.

A parser is an algorithm that attempts to determine whether a given input sentence can be derived from a nominated nonterminal symbol (the ‘start symbol’). The evidence of a successful parse can be a parse tree effectively encoding the steps of a derivation process. For a more extensive take on grammars and parsing, the reader is referred to [3, 79].

In the context of software languages, ambiguities in a grammar are often considered as flaws of the grammar introduced by the *grammar engineer* who wrote the grammar.² Many examples of ambiguities in real-world software language definitions exist [217], e.g., in ANSI-C, and ambiguities can be notoriously difficult to detect [18]. An ambiguity can be resolved by a refactoring of the grammar that preserves the set of sentences generated by the grammar. Additional refactorings, such as left-factoring, can be performed to reduce the non-determinism of the grammar. Such refactorings are often performed, explicitly or implicitly, by a *parser engineer* responsible for implementing a parser for the language. The implemented parser should be sound with respect to the (original) grammar definition such that it accepts only derivable sentences. This is especially the case when the grammar forms a contract between the grammar engineer and the *programmer*, e.g., when the grammar is part of a reference manual. In this case, helpful error messages provided by a parser can refer to the grammar to help solve syntax errors in a program. Figure 6.3 visualizes the user roles, artifacts and relations in this (idealized) view on syntax and parsing.

¹ Strictly speaking, a grammar is ambiguous iff there are multiple *left-most* or *right-most* derivations of a sentence.

² Although the developers of software language workbenches typically accept ambiguity as a consequence of a more ‘natural’ definition of a language and introduce ambiguity reduction annotations to the grammar formalism provided by the workbench to disambiguate under-the-hood.

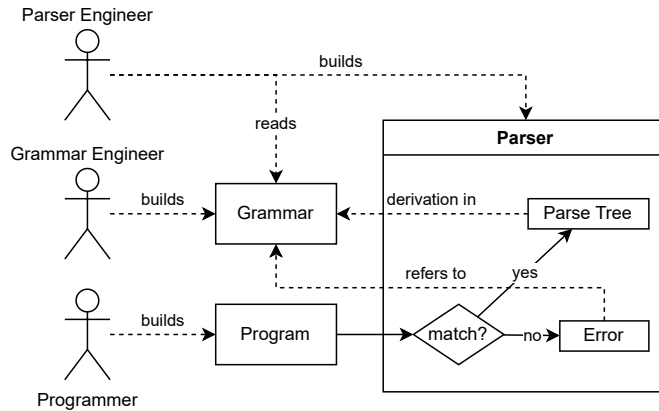


Figure 6.3: The ideal relation between the different roles and artifacts involved in the production and usage of grammars and parsers. The program is the input sentence to the parser.

Based on this view, we have defined four user stories for the various user roles.

1. As a grammar engineer, I want to diagnose and remove ambiguities in the grammar.
2. As a grammar engineer, I want to explore the set of sentences (the language) generated by the grammar.
3. As a parser engineer, I want to discover whether the grammar is in the class $LL(1)$, as this enables (hand-written) deterministic recursive descent parsers.
4. As a programmer, I want to better understand syntax errors reported to me by a (deterministic or non-deterministic) top-down parser.

In the next subsection, we will use the multiverse debugging framework to define a debugger targeting these user stories. The STR of this debugger captures the derivation process as described at the start of this subsection. Top-down and recursive descent parsers (such as $LL(k)$ and GLL) implement an algorithm that resembles this process quite closely. As a result, a debugger that simultaneously aids engineers of grammars and top-down parsers is more likely (to be feasible and intuitive) than a debugger that combines diagnosis for grammars and bottom-up parsers (such as $LR(k)$ and LALR). We have left bottom-up parsing out of scope in this chapter.

6.3.1 An STR for Grammars

The STR interface for debugging grammars is defined as follows. A configuration is defined as a tuple $\langle S, i \rangle$, where S is a stack and $i \geq 0$ is an index into some input sentence I . The input sentence I and the grammar \mathcal{G} do not occur in a configuration as they are constant per derivation/parsing session. The configuration $\langle (\mathcal{X} \rightarrow \cdot \mathcal{G}_S), 0 \rangle$ is the single starting configuration in which \mathcal{G}_S denotes the nominated start symbol of the grammar and \mathcal{X} a fresh, auxiliary start symbol used to detect a successful derivation (see Rule ACCEPT in Figure 6.4). The elements of the stack are quadruples displayed as $(X \rightarrow \alpha \cdot \beta, i)$, with i an index, $(X, \alpha\beta)$ a production, α, β are sequences of symbols, and \cdot denoting the progress made in matching the production, with the symbols α already matched. Notationally, the stack is a sequence of elements separated by the \bullet symbol, with the more recently pushed element on the right. We refer to the symbol after the \cdot in the top-most element of the stack as the next symbol to match.

The actions are given by the following grammar:

$$a \in \text{actions} ::= \text{match}(t) \mid \text{descend}(X, \alpha) \mid \text{ascend} \mid \text{accept}$$

Meta-variable t refers to a token, X to a non-terminal symbol, and α to a sequence of symbols. The semantics of the actions are given as inference rules in Figure 6.4. The $\text{match}(t)$ action is available when terminal t is the next symbol to match and when the current index i points to t in the input sentence. The $\text{descend}(X, \alpha)$ action is available when nonterminal X is the next symbol to match and when $X \rightarrow \alpha$ is a production in the grammar. The descend action corresponds to the function call of a recursive descent parser in which the stack resembles the call-stack of the parser. The ascend action is available when there is no next symbol to match and is analogous to returning from a call to a recursive descent parser. The accept action is available only when the stack indicates the start symbol of the grammar has been matched and the end of the input has been reached, indicating a completed derivation/parsing process. The naming of the actions is inspired by the characterization of recursive descent (top-down) parsing algorithms given in [25].

6.3.2 User Interactions

A debugger is considered to satisfy a user story if it affords a sequence of user interactions that together (sufficiently) support the user in realizing the goal formulated in the user story. In the analysis we focus on a theoretical/objective realization of the story rather than user experience. That is, we determine whether the sequence of interactions

$$\begin{array}{c}
\frac{I_i = t}{\langle S \bullet (X \rightarrow \alpha \cdot t\beta, j), i \rangle \xrightarrow{\text{match}(t)} \langle S \bullet (X \rightarrow \alpha t \cdot \beta, j), i + 1 \rangle} \quad (\text{MATCH}) \\
\frac{S = S' \bullet (Y \rightarrow \alpha \cdot X\beta, j) \quad (X, \delta) \in \mathfrak{G}}{\langle S, i \rangle \xrightarrow{\text{descend}(X, \delta)} \langle S \bullet (X \rightarrow \cdot \delta, i), i \rangle} \quad (\text{DESCEND}) \\
\frac{S' = S \bullet (Y \rightarrow \alpha X \cdot \beta, k)}{\langle S \bullet (Y \rightarrow \alpha \cdot X\beta, k) \bullet (X \rightarrow \gamma \cdot, j), i \rangle \xrightarrow{\text{ascend}} \langle S', i \rangle} \quad (\text{ASCEND}) \\
\frac{I_i = \$ \quad S = (X \rightarrow \mathfrak{G}_S \cdot, 0)}{\langle S, i \rangle \xrightarrow{\text{accept}} \langle S, i \rangle} \quad I(c, a) = \{c' \mid c \xRightarrow{a} c'\} \\
\quad (\text{ACCEPT}) \quad \quad \quad (\text{Interpreter})
\end{array}$$

Figure 6.4: Semantics of the grammar-engineering domain-specific debug actions, and the resulting interpreter. For simplicity, the rules encode LL(1). LL(k) can be obtained with slight modifications to the handling of indices in the rules.

yields the informational content needed for the diagnosis, not on how this information is made available or presented.

To explore suitable interactions for the ambiguity user story, let i , j be integers, and X a nonterminal. All configurations of the form $\langle S \bullet (X \rightarrow \gamma \cdot, j), i \rangle$ in the reachability graph generated by the interpreter indicate that the nonterminal X can derive the subsentence $I_{j,i}$ of the input I ranging from j to i . The grammar engineer can inspect the reachability graph in two ways: (1) choosing a concrete γ and finding multiple paths in the reachability graph reaching the corresponding configuration and (2) finding multiple configurations for different choices of γ . In both cases, known as horizontal and vertical ambiguity respectively [18], there is a (left-most) derivation of $I_{j,i}$ per path and the path gives the specific and concrete details of the derivation. Analyzing and comparing these paths gives insight into the nature of the ambiguity, which can then be used to resolve the ambiguity by modifying the grammar. A feasible reduction discards the tail of the stack (S above) from a configuration. Figure 6.8 gives several examples of breakpoint and reduction expression for this case study. Based on these observations, we define the following two requirements.

Requirement 1 (RQ1)

A debug user should be able to define breakpoints over paths.

Requirement 2 (RQ2)

A debug user should be able to find all configurations satisfying a breakpoint

The first requirement supports the first case of ambiguity, where a user can use a breakpoint to find a configuration with multiple incoming edges. The second requirement supports the second case of ambiguity, where a user can find all configurations that project the same information (X, i and j in this case).

A grammar is left recursive when it contains a production rule $X \rightarrow \alpha$ for which holds that the derivation process applied to α can yield a sentence of the form $X\gamma$, for some γ , via one or more derivation steps. The reachability graph generated by the interpreter applied to a left-recursive grammar is infinite as infinitely many descend actions on X can be performed whilst ever-growing the stack. To still utilize breakpoint finding in such scenarios, a bounded search can be applied. Hence, we formulate the following requirement.

Requirement 3 (RQ3)

A debug user should be able to control the depth of breakpoint searches.

Instead of indiscriminately bounding the search space, we can also filter out only those paths where the stack displays more ‘recursive calls’ than elements left in the input sentence (an approach to handle left-recursion suggested in [74]). We therefore formulate the following requirement.

Requirement 4 (RQ4)

A debug user should be able to preemptively reduce (prune) the search space.

The second user story is related to sentence generation. The STR can be used to generate sentences by (selectively) ignoring I (the input string) when determining whether the *match* and *accept* actions are available actions to execute. Paths in the reachability graph ending with an *accept* transition then display sentences derivable from the start symbol (by inspecting the *match*-transitions of the path). Note

that there may be infinitely many such paths. A grammar engineer might be interested in sentences of a particular structure by selectively disabling the input sentence. The order of the sentences generated via *find* is determined by the implemented search strategy. The existing framework enforces a depth-first search strategy, which may not yield a representative sample of the sentences of the grammar when applied a limited amount of times. This brings us to formulate the following requirement.

Requirement 5 (RQ5)

A debug user should be able to control the search strategy used during breakpoint finding.

The third user story is related to the implementation of the parser. If the grammar is in the class of $LL(1)$ grammars, the parser engineer can hand-write a performant recursive descent parser. We modify the debugger by adding a condition that makes an action $descend(X, \alpha)$ available only if I_i is in the *first-set* of α (first-sets can be precomputed from the grammar definition [79]). A concrete counter-example to the $LL(1)$ property is found if there is a configuration admitting two or more *descend* actions. Requirements 1, 2, 3 suffice.

The fourth and final user story is related to a programmer interacting with a parser: The programmer wants to better understand the error they made in a program rejected by a parser. To achieve this, the programmer can use the debugger by searching for configurations not admitting any actions (the derivation cannot continue and is not complete). The programmer can then inspect the trace of the current execution to obtain information regarding the parse process and possibly relate this to the parse error.

Since our focus is on top-down parsers, which typically perform $LL(1)$ -lookahead, the first-set condition described above would find states without outgoing actions earlier in the search, therefore reducing the work requires during the manual or automatic search. If the parser is deterministic, and the grammar is $LL(1)$, then our debugger will run into the same unique error. If the parser is non-deterministic, we can use the debugger to find all error states discovered by the parser. Basic breakpoints and Requirement 2 suffice.

6.4 FUNCONS

The funcon framework has been a foundational aspect to parts of this thesis. In Chapter 3 and Chapter 4, we used funcons as the semantic domain for our defined languages. Debuggers for the defined languages

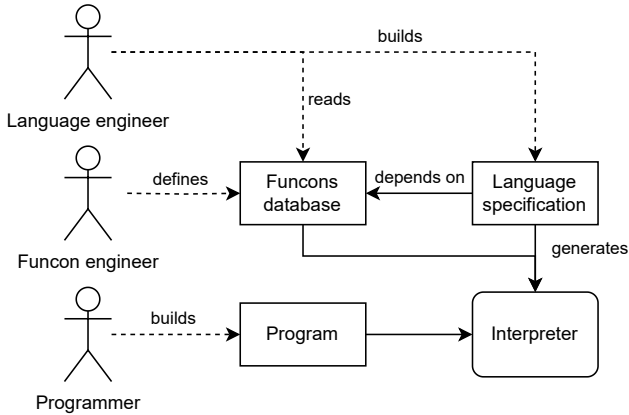


Figure 6.5: The ideal relation between the different roles and artifacts involved in the production and usage of operational semantics and (derived) interpreters.

can be defined in terms of a funcon debugger. In this case study we explore such a funcons debugger.

Every funcon is formally defined by a *funcon engineer* in the CBS meta-language [144], using small-step I-MSOS [143, 146], a modular variant of structural operational semantics (SOS) [164]. The operational semantics of a programming language is defined by a *language engineer* via a translation from object language programs to funcon terms, as demonstrated with *iCoLa* and *iCoLa⁺* via semantic functions. A *programmer* (language user) writes a program in the object language as input to an interpreter that first converts the input program to a funcon term (following the translational semantics) and then applies the existing funcon term interpreter to the resulting funcon term [28]. In this case study we investigate whether and how the multiverse debugging framework can be used to obtain a multiverse debugger for both funcon terms and object language programs. Figure 6.5 visualizes the aforementioned user roles, artifacts and relations in this (idealized) view on reusable programming language semantics.

The funcon term interpreter is directly derived from the small-step definition of the funcons written in CBS [28]. The small-step nature of these definitions makes it possible to instrument the funcon interpreter to enable stepwise debugging, with each step in the underlying SOS semantics corresponding to a step in the debugger.³ The funcon interpreter is non-deterministic if at least one of the funcons in the funcon library is non-deterministic, which is the case if: The funcon

³ Note that this process may not result in the desired granularity of steps.

is defined by (I-MSOS) rules that are not mutually exclusive, i.e., two or more rule instances⁴ can be simultaneously applied to perform a step on a given funcon term. Or, the funcon term is given an informal or axiomatic definition that is inherently non-deterministic. The first source of non-determinism is generally considered to be undesirable when defining the semantics of a deterministic or sequential programming language. However, one can use this source of non-determinism to specify the behavior of concurrent programming constructs (e.g., threads) and non-deterministic operators (e.g., without a well-defined order of argument evaluation).

In the current funcon library, `set-elements` is a non-deterministic operator which returns a permutation of the elements of a set. Crucially, the order of the returned sequence is unspecified. This funcon is directly or indirectly used in the definition of other non-deterministic funcons such as `some-element` (yielding an arbitrary element from a non-empty set), which is used to define the semantics of concurrent programming languages based on the thread-model.

Based on the aforementioned described domain, we have defined five user stories.

1. As a funcons engineer, I want to introduce new (non-deterministic) funcons and explore their semantics and interaction with existing funcons.
2. As a language engineer, I want to experiment with funcons to determine the right combination of funcons for my language semantics.
3. As a language user, I want to query the current program state.
4. As a language user, I want to see the code around the current program point.
5. As a language user, I want to modify the program and observe the effects of the modification.

6.4.1 *A STR for Funcons*

The STR configuration for funcons consists out of the current (funcon) term under execution and a set of auxiliary semantic entities capturing contextual information such as variable bindings, assignments and printed output (see [28]). The funcon debugger has one action: `step`. Its semantics and the resulting interpreter are as follows.

⁴ In SOS and variants, a rule is instantiated to form a rule instance by substituting meta-variables, not all of which may be bound by the term under evaluation, creating a source of non-determinism.

$$\frac{t \xrightarrow{(e,e')} t'}{\langle t, e \rangle \xrightarrow{\text{step}} \langle t', e' \rangle} \quad (\text{STEP}) \qquad I(c, a) = \{c' \mid c \xrightarrow{a} c'\} \quad (\text{Interpreter})$$

The step action steps the current term and updates the configuration with the derived term and the updated entities, if any. If the step triggers the evaluation of a non-deterministic funcon, the step may yield more than one output configuration.

6.4.2 User Interactions

The first user story is from a funcon engineer's viewpoint who wants to explore the right definition for a non-deterministic funcon. A funcon is non-deterministic if the reachability graph contains a configuration with multiple outgoing edges. Such configurations can be easily found using Requirement 1. However, even with a finitely branching small-step transition system, one step can introduce many new configurations. This can happen either when the root term directly produces many result configurations, or because sub-computations produce many result configurations, which can aggregate. So, performing one step can still cause an enormous growth of the state space, therefore making it infeasible to control manually. Nevertheless, not all the non-determinism observed during a step is of interest. Therefore, by focusing only on non-determinism that is significant to the debugging task at hand, the amount of states produced by one step can be significantly reduced. We therefore formulate the following requirement.

Requirement 6 (RQ6)

A debug user should be able to control for which non-deterministic terms all states are visited.

The second user story is from the perspective of a language engineer who uses funcons to give semantics to an object language. This type of user might utilize the debugger to explore semantics for a specific language construct by giving a definition for that construct in terms of funcons and testing the construct. The language engineer can achieve this by starting a new debugging session for every example program and observing the behavior. However, that makes it difficult to compare debug sessions, which is a primary goal in this user story. With that in mind, we define the following requirement.

Requirement 7 (RQ7)

A debug user should be able to go back-in-time to retry a scenario with different input values and compare the outcome values of the different scenarios.

The third and fourth user stories are from the perspective of a programmer using an object-language with semantics defined in terms of funcons. Both these user stories describe interactions with the debugger that enable a programmer to get a better idea of the current state the program is in, without getting overloaded with too much information. The third user story does this by enabling the user to query specific information out of a specific state. And the fourth user story enables a user to build a better mental model where the execution is paused. Both of these interactions can be achieved via the breakpoint and reduction functionality already present in the debugger. A query on the state can be formulated as a breakpoint, if the answer is boolean. Alternatively, a reduction and display of the reduced configuration can project specific information out of configurations. We therefore formulate the following requirement.

Requirement 8 (RQ8)

A debug user should be able to test for breakpoints and to visualize (reduced) configurations.

6.5 eFLINT (REASONING WITH NORMS)

In Chapter 2 we utilized eFLINT as a case study for the work on exploratory programming. In this section, we use eFLINT as a case study for the work on multiverse debugging. As mentioned in the introduction, exploratory programming and multiverse debugging have some commonalities but operate at a different granularity. By utilizing eFLINT as a case study for both, the differences and commonalities become more apparent.

The eFLINT language is a domain-specific language for reasoning with formalized interpretations of norms as they are found in laws, regulations, contracts and organizational policies [27]. In eFLINT, a normative specification encodes a formal interpretation of norms, declares data-structures (types) for knowledge representation whose instances (facts) are either true or false. This part of a specification is referred to as the *ontology* of the specification. In the *process model* of a specification, effects are associated with action- and event-types, determining which facts are rendered true or false by the triggering of instances of these

types (actions and events, respectively). Together, the ontology and the process model describe a finite, non-deterministic state automaton in which states are formed by the set of (true) facts and transitions are formed by the (effects of) actions and events. The automaton is non-deterministic in that in any given state, multiple actions and/or events may be triggerable. The automaton is finite as there is a finite amount of (possible true) facts and because the amount of outgoing transitions of any state is bounded by the finite set of possible actions and events.

The *normative classification*, the final part of an eFLINT specification, assigns violations to states and transitions of the automaton⁵. A duty-type declaration establishes that a state is violating if it states the truth of an instance of the type (a duty) whilst also satisfying one or more of the violation conditions (Boolean expressions over facts) associated with the duty-type. An action-type declaration establishes that a transition is violating if one or more of the pre-conditions (Boolean expressions over facts) associated with the action-type is not satisfied by the source-state of the transition.

The language can be used to establish the extent to which a software system complies with (the formalized interpretation of norms encoded in) a policy document. In an idealized setting, visualized in Figure 6.6, a *policy expert* determines the policy – possibly including relevant laws and regulations – of an organization employing some software system. Following a model-driven approach, a *software engineer* maintains⁶ both the running software system as well as the parts of the eFLINT specification that model the software system (ontology and process model). The *policy engineer* extends this eFLINT specification with the normative classification, established by formalizing the norms encoded in the policy.⁷ Based on the above description of the domain, we formulate the following five user stories.

1. As a policy engineer, I want to discover in what ways particular transitions or states can be reached.
2. As a policy engineer, I want to discover the effects on possible violations of certain modifications to the normative classification of an eFLINT specification.

⁵ In practice, a normative classification will also introduce sets of *institutional* facts and actions that play a role in establishing compliance, separate from the *physical* facts and actions representing the software system, see [27, 61, 152, 194].

⁶ We are not concerned here with whether or how one is derived from the other or how the two are kept consistent.

⁷ We are also not concerned with the processes required to integrate concepts from the policy and software system

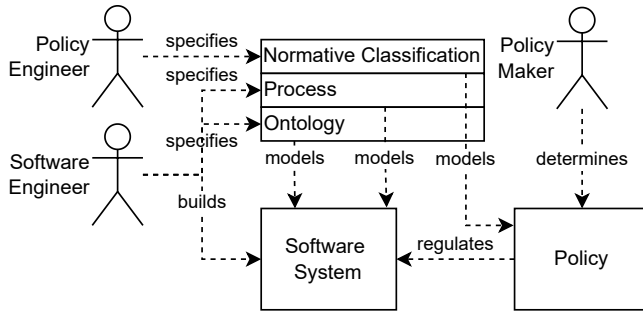


Figure 6.6: The ideal relation between the different roles and artifacts involved in checking the compliance of a software system against policy using eFLINT.

3. As a policy engineer, I want discover how to modify the normative classification to ensure certain states or transitions are (no longer) violating.
4. As a software engineer, I want to assess the compliance risks of the software system modeled by an eFLINT specification by determining in what ways violating states and transitions can be reached.
5. As a software engineer, I want to modify the process model of an eFLINT specification to reduce the number of possible occurrences of violations.

6.5.1 A STR for eFLINT

The eFLINT STR consists of a configuration defined as a tuple containing the current specification and the current knowledge base. eFLINT has one type of action: *trigger t*. A *trigger* action is generated for every trigger-able action in the knowledge base. The semantics of the debugging action and the resulting interpreter are as follows.

$$\frac{t \in kb \quad kb \xrightarrow{t} kb'}{kb \xrightarrow{\text{trigger } t} kb'} \text{ (TRIGGER)} \qquad I(c, a) = \{c' \mid c \xrightarrow{a} c'\} \text{ (Interpreter)}$$

From this definition we observe that a debugging session is fixed over a specification that determines the explorable search space. With exploratory programming the specification is not fixed but evolves throughout a session. This ties in directly with the aforementioned granularity at which the two methods operate.

6.5.2 *User Interactions*

For the first user story, the user wants to find states that are reached in a particular way. This user story is already captured by Requirement 1.

The second user story is focused on comparing paths or configurations at different points in a debugging session. We therefore formulate the following requirement.

Requirement 9 (RQ9)

A debug user should be able to inspect partial debug traces.

This requirement differs from Requirement 7 in two aspects: no back-in-time functionality is required, and only one particular trace is inspected.

The third user story is concerned with assessing the effects of modifications to the normative specification, such as the possible violations that can occur in a system. This can be achieved by altering the specification being debugged and re-evaluating a particular scenario. The requirement for such interactions correspond to Requirement 7.

The fourth user story concerns finding compliance related breakpoints for a scenario in a specification. A user could achieve this by defining a breakpoint that finds states in which a violation exists. To find multiple such points, the debugger needs to support finding multiple breakpoints, as formulated in Requirement 2. To also determine how these different breakpoints were reached, the debugger needs to keep track of multiple histories and make multiple histories insightful. Hence, we formulate the following requirement.

Requirement 10 (RQ10)

A debug users should be able to operate on multiple histories in one debug session.

The fifth user story concerns the viewpoint of a systems engineer that wants to reduce violations in their system by modifying the process. This user story is similar to the third user story, but from the viewpoint of a different actor. Nevertheless, the specific user interaction is the same. Hence, the requirements needed to satisfy the current user interactions are also captured by Requirement 7.

6.6 GENERALIZED MULTIVERSE DEBUGGING

We introduce the generalized STR (GSTR) which adapts the STR-based debugging framework along three dimensions: new components for meta-actions are added to the STR, the history and options component are generalized using graph structures instead of sets, and the breakpoint (B) and reduce (R) functions are generalized by *Step* and *Label* functions.

The meta-action components are motivated by Requirements 7,8, and the removal of the breakpoint and reduce functions. Using meta-actions, language engineers can extend their debugger with language-specific functionality. This is extra useful for visualization and query operations which do not alter the configuration in any way, but as actions would still be included in the history. With meta-actions, the functionality remains without polluting the history.

Using graphs instead of sets for the history and options component is motivated by Requirements 1,9. With graphs, more information is retained, empowering more expressive breakpoint and reduction functions.

The last adaptation generalizes the breakpoint and reduce functions by *Step* and *Label* functions, and is motivated by Requirements 2,3,4,5. The *Label* function assigns a label to every configuration in the current search graph. The *Step* function performs a step on the current search graph based on the labeling by the *Label* function. After a step, the search graph is extended with new configurations and another iteration of labeling and stepping is performed. When no new configurations are added the algorithm stops. Compared to the original breakpoint (B) and reduce (R) functions, the *Step* and *Label* functions provide more flexibility and expressiveness, while also promoting reusability among different language-specific debuggers. For instance, in our implementation we have defined several reusable functions that can be combined to create concrete *Step* and *Label* instances. It is thus possible to implement new search strategies without modifications to the debugging framework.

6.6.1 Formal Generalized-STR Definition

We now give the formal definition of the GSTR, following the formal definition in Section 6.2.

Definition 6.6.1. A generalized STR (GSTR) is a tuple $\langle C, C_0, A, M, I, Act, P, Com \rangle$, which extends the STR with three new elements: M, P, Com . M denotes a set of meta-actions (or commands). P is a function $C \times M \rightarrow C \times O$ which performs a command on a con-

figuration, resulting in an updated configuration and an output value. The set O is left abstract but is defined by the debugging framework and varies depending on the execution environment. It provides a mechanism for meta-actions to communicate with the external world.⁸ Finally, the Com component is a function $C \rightarrow \mathcal{P}(M)$ giving the active commands for the given configuration.

A generalized debugger is defined in terms of a GSTR, where $GS = \langle C_s, C_{s0}, A_s, M_s, I_s, Act_s, P_s, Com_s \rangle$ is the GSTR for the language being debugged:

$$GD_{GS}(Step, Label) = \langle C_d, C_{d0}, A_d, M_d, I_d, Act_d, P_d, Com_d \rangle.$$

The debugger configuration is defined as a tuple:

$$C_d = \langle C_s \cup \{\perp\}, \mathcal{G}(C_s, A_s), \mathcal{G}(C_s, A_s) \rangle,$$

with $\mathcal{G}(C, A) = \langle \mathcal{P}(C), \mathcal{P}(C \times A \times C) \rangle$ denoting a graph with vertices being elements of C and edges are labeled by elements of A . The set of actions A_d is the set of actions of STR debugger extended with a *meta* m action for execution of the meta-commands in M_s . For the debugger, we leave the set of meta-commands (M_d) empty. Therefore, the function P_d is a constant function returning the given configuration and no output. The debugger is indexed by two functions: *Step* and *Label*, which generalize over the *Breakpoint* (B) and *Reduce* (R) functions from the STR definition, which is discussed in more detail in §6.6.2.

$$Step : (C \rightarrow L) \rightarrow \mathcal{G}(C, A) \rightarrow \mathcal{G}(C, A)$$

$$Label : \mathcal{G}(C, A) \rightarrow (C \rightarrow L)$$

The set L is a label set with elements forming labels. Every label set comes associated with two functions $\langle accept, enabled \rangle$ of type $L \rightarrow \mathbb{B}$. The *accept* function denotes whether a particular label indicates that the associated configuration is an accepting state. The *enabled* function denotes whether a particular label indicates that the associated configuration is enabled for transitions. The *Step* function iterates the graph in such a way that only new outgoing edges are added to vertices with a label marked as *enabled*. The debugger itself uses the *accept* function to extract the *interesting* configurations after a search. The iterative process is performed until a fixed-point is reached, which requires that the *Step* function is monotonic on the structure of the graph. Finally, the semantics of the debugger (I_d) is updated, shown in Figure 6.7.

⁸ An alternative method to model external communication is to execute the meta-action in a monad. For simplicity, we have opted to model it using an abstract output value.

$$\begin{array}{c}
\frac{a_s \in Act_s(c_s) \quad c_s \neq \perp \quad cs = I_s(c_s, a_s)}{opts = (cs, \{(c, a, c') \mid c' \in cs\})} \quad \text{(STEP)} \\
\frac{\langle c_s, hist, _ \rangle \xrightarrow{step\ a_s} \langle \perp, hist, opts \rangle}{c_s \in \mathcal{V}(opts) \quad g' = g \cup_G opts} \quad \text{(SELECT)} \\
\frac{\langle _ , g, opts \rangle \xrightarrow{select\ c_s} \langle c_s, g', \epsilon \rangle}{c_s \in \mathcal{V}(g)} \quad \text{(JUMP)} \\
\frac{\langle _ , g, _ \rangle \xrightarrow{jump\ c_s} \langle c_s, g, \epsilon \rangle}{c_s \neq \perp \quad m_s \in Com_s(c_s) \quad M_s(m_s) = \langle c', o \rangle} \quad \text{(META)} \\
\frac{\langle c_s, g, \epsilon \rangle \xrightarrow{meta\ m} \langle c', g \frown c', \epsilon \rangle}{find_\psi(\{\{c_s\}, \emptyset\}) = (gn, cn)} \quad \text{(RUN)} \\
\frac{\langle c_s, g, _ \rangle \xrightarrow{run_to_breakpoint} \langle \perp, g \cup_G gn, cn \rangle}{find_\psi(opts) = (gn, cn)} \quad \text{(RUN-2)} \\
\frac{\langle \perp, g, opts \rangle \xrightarrow{run_to_breakpoint} \langle \perp, g \cup_G gn, cn \rangle}{I_d(c_d, a_d) = \{c'_d \mid c_d \xrightarrow{a} c'_d\}} \quad \text{(Interpreter)}
\end{array}$$

Figure 6.7: Semantics of the debugging operations for the generalized debugger. The function \mathcal{V} projects the vertices out of a graph. The \frown operation adds a configuration to the vertices of a graph. The \cup_g operation combines two graphs by taking the pointwise union. We use ϵ for empty graphs.

Compared to the STR definition, the GSTR definition does not use *Break* and *Reduce* functions. Instead, *Step* and *Label* functions are used, and the history and options components are now generalized to graphs. This generalization is the biggest contributor to the changes required in the semantics of the debug actions.

6.6.2 Concrete Step and Label components

To show the expressiveness of the *Step* and *Label* functions, we highlight some of the breakpoint and reduction expressions from our grammar case study, and explain how to obtain the functionality of the original breakpoint and reduce functions using *Step* and *Label* functions.

Figure 6.8 highlights several breakpoint and reduction expressions defined for our grammar case study. Several of these examples were discussed from the user-interaction point-of-view in §6.3.2. The first breakpoint finds configurations denoting a successful parsing derivation. For this breakpoint, the *Label* function checks for every configuration if the condition is met, and if so the configuration is labeled as accepting. The second breakpoint finds configuration for which there exist two unique paths to some other configuration. The first reduction reduces configurations in which the next terminal to match is not a member of the computed follow-set. The second reduction reduces configurations where the stack is larger than the size of the input not yet matched. Both reductions do not work with a seen set, and instead prune, by labeling the configuration disabled, the search space immediately when a configuration that satisfies the expressions is found.

To obtain the breakpoint/reduce functionality from the STR-debugger, we define a label set with three labels: *enabled*, *disabled*, and *accepted*. The *Label* function maps the graph to a reduced graph according to some reduction function, and assigns labels to the configurations according to the original semantics of the STR-based debugger: *accepted* if a configuration matches the breakpoint, *enabled* if a reduced configuration has no outgoing edges, and *disabled* otherwise. The *Step* function does a depth-first search until there exists a configuration with an *accepted* label or until there exists no configuration with an *enabled* label. The implementation is parametrized by the reduction and breakpoint function. This parametrization is fully hidden from the debugging framework.

6.7 SATISFACTION OF THE REQUIREMENTS

Table 6.1 discusses for each framework whether it satisfies a requirement and if not which modifications can be made to the framework to satisfy the requirement.

Breakpoints:

$$(\forall c)(\exists c')(c \xrightarrow{\text{accept}} c'). \quad (\text{Accepting states})$$

$$(\forall c)(\exists c', p, p')(c' \xrightarrow{p}_m c \wedge c' \xrightarrow{p'}_n c \wedge p \neq p'). \quad (\text{Ambiguity points})$$

Reductions:

$$(\forall c)(\mathcal{I}[c.i] \notin \mathcal{F}(c)).$$

$$(\forall c)(\exists X, \alpha)(\text{count}(\text{descend}(X, a), c.S) > \text{length}(\mathcal{I}) - c.i).$$

Figure 6.8: Example breakpoint and reductions applicable to the GSTR of the grammar-engineering case study. We use $\mathcal{F}(c)$ to denote the follow-set. Traces longer than 1 step are subscripted to denote the length of the trace.

At a high level, the first five requirements are met by our framework due to the introduction of the *Step* and *Label* functions. Some of the requirements can be met by the original framework via a small modification of the semantics, for example via an alternative implementation of the *find* function. This is also what we observed during the implementation of the case-study debuggers. Based on these observations, we came to the generalization via the *Step* and *Label* components that encompass all those requirements while also offering reusability and flexibility between different debugger implementations. With the *Step* and *Label* components, new search strategies, essentially alternative implementations for the *find* function, can be defined without needing modifications to the debugging framework semantics. Therefore, a *Language engineer* is not dependent on a *Framework engineer* when desiring alternative search strategies.

The second set of the requirements is more focused on the history mechanism, and most requirements are met by both frameworks. Still, the introduction of the graph-based history adds several new possibilities to the debuggers, including the generalization of the reduce and breakpoint functionality, while also supporting multiple independent debugging explorations in the same debugging session.

Finally, Requirement 6 is met by both frameworks, but not using a reusable mechanism. Instead, the required work is pushed to the interpreter of the language being debugged. Satisfying this requirement in a reusable manner requires interaction between the debugger and interpreter on every sub-computation, which requires severe alterations to the interpreter implementations. By not integrating this, we keep

the framework interface simpler for languages that do not need the support for this feature, while still supporting the feature for languages that require it.

Table 6.1: Analysis of the extent to which the debugging framework of [156] and our extended version satisfy the requirements formulated for the case studies of this chapter.

Requirement	Pasquier et al. [156]	Our work
RQ ₁	This requirement is met using a modified <i>find</i> function as demonstrated by [157].	This requirement is met owing to the introduction of the graph history and the <i>Label</i> function over this history, which can be defined such that it assigns a breakpoint label to configurations based on paths in the graph.
RQ ₂	This requirement can be met via a modification to the <i>find</i> function of the debugging framework, which currently performs a depth-first search and stops after finding a breakpoint.	This requirement is met owing to the definition of the <i>find</i> function in terms of the <i>Step</i> and <i>Label</i> functions. The <i>Step</i> and <i>Label</i> functions can be defined such that they continue the search until <i>all</i> breakpoints are reached. Termination of this process depends on the concrete <i>Label</i> function, and the underlying language being debugged.
RQ ₃	This requirement is not met due to the fixed semantics of the <i>find</i> function, which continues until either all (reduced) configurations have been visited or a breakpoint is reached. Nevertheless, this requirement is easily satisfied by modifying the <i>find</i> function to take an integer parameter to control the recursion depth of the search.	This requirement is met by defining a <i>Label</i> function that disables all configurations when there exists a non-repeating path in the graph of certain length. By disabling all configurations, the <i>Step</i> function cannot progress on any configuration, and the search will be terminated.

RQ4	This requirement can be met via a modification to the <i>find</i> function of the debugging framework, which currently utilizes a set of previously <i>seen</i> (reduced) configurations to handle pruning of the search space.	This requirement is met by defining a <i>Label</i> function that performs pruning based on properties of configurations in the graph. The previously <i>seen</i> (reduced) configurations is an example of such a property, but other implementations are possible, such as the pruning of left-recursion in the grammar case study.
RQ5	This requirement can be met by modifying the <i>find</i> function of the debugging framework, which currently performs a depth-first search.	This requirement is met by defining <i>Step</i> functions with different search strategies. For our case studies, we implemented depth- and breadth-first search. Other strategies, such as a parallel search strategy, are also possible.
RQ6	This requirement is met, but not in a reusable manner. Instead, the language designer needs to encode this functionality in the interpreter and use the configuration to communicate which non-deterministic terms need to be collapsed and which terms need to be fully explored.	This requirement is met, but not in a reusable manner. Instead, the language designer needs to encode this functionality in the interpreter and use the configuration to communicate which non-deterministic terms need to be collapsed and which terms need to be fully explored.
RQ7	This requirement is met via the support of user-definable actions and <i>jump</i> . A language engineer can add an action that modifies the program being debugged. A user can then <i>jump</i> to the specific configuration and perform the modification action at that point.	This requirement is met via the support of user-definable actions and <i>jump</i> . A language engineer can add an action that modifies the program being debugged. Meta-actions can be used instead, adding an isolated configuration to the history, resulting in a clearer divide between different explorations in the same debugging session.

RQ8	This requirement is not met by the framework because breakpoints and reductions are purely available inside the <i>find</i> function. Nevertheless, it would be trivial to extend the framework with this semantics by adding two new actions to the debugger, one to test a breakpoint on the current configuration, and one to reduce the current configuration.	This requirement is met by our framework via the usage of meta-actions in combination with the output result. Direct support from the framework for this requirement is thus removed. Requiring the usage of meta-actions to satisfy this requirement does move some implementation efforts away from the framework to the language engineer.
RQ9	This requirement can be satisfied through a relatively simple modification to the framework, using a tree or list to represent history instead (also discussed in [156]).	This requirement is satisfied owing to storing the history as a graph, which makes it trivial to focus on (partial) traces of the current debugging session.
RQ10	This requirement is satisfied via the <i>jump</i> action, which makes it possible to go back to a previous configuration and explore a different part of the history, thus supporting multiple histories in one debugging session.	This requirement is satisfied via the <i>jump</i> action, which makes it possible to go back to a previous configuration and explore a different part of the history, thus supporting multiple histories in one debugging session.

6.8 EXPLORATORY PROGRAMMING AND MULTIVERSE DEBUGGING

By fulfilling Requirement 7, multiverse debugging moves into the direction of exploratory programming. Nevertheless, there is a still difference between the two and the two approaches complement each other. We can see the support for exploratory programming like functionality in a multiverse debugger as a way to evolve a debugging session alongside an exploratory programming session. Every configuration in an exploratory programming session gives rise to a new initial configuration for a multiverse debugging session. With the functionality fulfilling Requirement 7, a new debugging session is not required for every configuration, instead they can be explored in the same debugging session, which supplements exploration performed during exploratory programming. Furthermore, one configuration in the exploratory session might be instantiated differently for different debugging scenarios. For example when defining a new programming language, where language variants corresponds to configurations in an exploratory session, a user might want to debug a program in a

language variant. Different programs being debugging for the same variant can then share a debugging session.

Alternatively, we can view the support for Requirement 7 as evolving an exploratory session alongside the debugging session, where new instantiations of debugging configurations cause evolution of an exploration session. This view is slightly more complex, since there must be a way to determine the programs that result in the initial debug configuration, and map these programs in a meaningful way to an exploration session.

6.9 THREATS TO VALIDITY

In this section we discuss the threats to validity present in this work from the point of view of empirical software research [50]. The primary component in our research is the requirements. The validity of the requirements can be affected by the chosen domains, and the selected user stories.

The selected user stories were determined by the authors based on expert experience in the respective domains. A more diverse set of user stories could be obtained by performing interviews with users across different experience levels. However, due to the fact that two out of the three domains have a small user base this was deemed impractical. In our case, we have thus opted to base our user stories on the experience of an expert in the respective domains.

To ensure our approach is transferable to different domains, we have performed our approach on three different domains. Furthermore, by being able to define the old framework in terms of the new framework, we retain the applicability of our work on those case studies.

6.10 RELATED WORK

The original multiverse debugging paper [125] presented Voyager, a multiverse debugger focused on AmbientTalk programs. As part of this implementation, they stored the exploration graph using the ArangoDB graph database. Hence, the graph of a debugging session can be queried using the ArangoDB query language. Our work is essentially a combination of the graph idea applied to the reusable framework of Pasquier et al. [157]. Although our implementation does not run on a (commercial) graph database, this is achievable in future work. Alternative options are also interesting, especially in combination with the *Step* and *Label* components. For example, using a graph algorithm language based on semigroups [101] or Kleene algebras [56] to guide the search. These ideas have been partially explored by [157] in the context of temporal breakpoints.

Omniscient debugging [121] enables back-in-time debugging, but does not have an explicit focus on non-deterministic programs. A reusable framework for omniscient debugging exists [33]. In addition, a significant amount of optimizations exist for omniscient debugging systems [17, 169]. In future work, it would be interesting to see if some of the optimizations can be applied to our framework, and how much work is required to extend existing omniscient debugging frameworks with multiverse debugging support.

With our debugger, a sequence of debugging interactions can be retried between different scenarios; a feature that is inspired by exploratory programming. In Chapter 2 we saw that using a graph as a history mechanism can introduce more traces than actively explored by the user. The full implications of this in the debugging context requires future work to determine. Nevertheless, our framework can be extended to keep a log of actions to reconstruct the concrete traces debugged by the user.

6.11 CONCLUSION

Debugging non-deterministic programs is challenging, primarily due to the state space explosion problem. Multiverse debugging aims to aid debugging such programs by making debugging an interactive and user-controlled exploration of the state space. Previous work introduced a reusable framework for multiverse debugging extended with user-definable reductions with which the search space can be reduced. With this framework, we have collected requirements for multiverse debuggers using three case studies. Based on these requirements we have identified several limitations in previous work. Using these insights, we have introduced a modified and extended framework for multiverse debuggers with more general applicability while promoting reusability, and making a clear connection with exploratory programming where multiverse debugging functions as a supplement to exploratory programming. The obtained framework for multiverse debugging was instantiated for funcons, resulting in a funcon term debugger. This debugger can be used by languages that have their semantics defined in terms of funcons, such as the languages defined in *iCoLa* and *iCoLa*⁺ that use funcons as their semantic domain.

CONCLUSION

Programming languages are a useful abstraction mechanism, demonstrated by the different languages defined and the abstractions they provide. These abstractions range from more concise syntax or different programming paradigms, with the goal of increasing productivity or maintainability, to enforcing domain constraints, with the goal of reducing bugs and increasing the resilience of software. However, designing and developing a new programming language when a suitable abstraction is identified — that is not yet sufficiently captured by an existing language — is complex. The engineering work to build a programming language is immense. Nevertheless, existing tooling, such as language workbenches, can reduce this engineering effort, making the construction of new programming languages as an abstraction mechanism a viable option. Although the engineering effort is reduced with such tooling, the design effort is not. Designing a programming language is complex, and making design decisions can affect the applicability or the ergonomics of the designed language. Correcting earlier design mistakes might require breaking changes, therefore requiring modifications to existing programs to be consistent with the new language. This is observed in many existing programming languages that have made (breaking) changes to their original designs.

In this thesis we have introduced an approach we call exploratory language development, which aims to aid the exploration of the design space by combining exploratory programming and language development into one activity. We captured this idea as a diagram in Figure 1.1, redisplayed here for convenience in Figure 7.1. The goal we set out in the introduction of this thesis, was to create an environment in which language developers can create languages and language variants without much effort, while being supported by exploratory programming. To experiment with these variants, switching between the meta-level — where languages are defined — and the object level — where defined languages are evaluated — needs to be easy. To further support the activity, both the meta-level and the object-level should have access to auxiliary tools.

7.1 CONTRIBUTIONS

To set a first step towards our vision, we have made several contributions that can be categorized into three categories: contributions to-

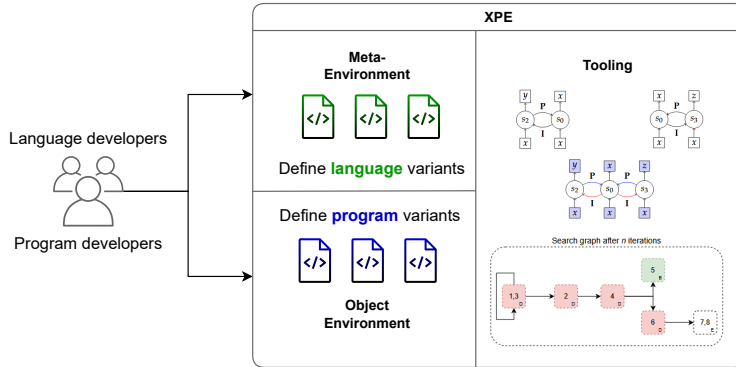


Figure 7.1: Conceptual overview of the activity of exploratory language development.

wards exploratory programming environments, contributions towards the meta-environment and object-environment, and contributions to tooling for program construction.

7.1.1 Exploratory Programming Environments

Exploratory programming environments support the activity of both language design and program construction using the designed languages. To easily obtain such environments, we have extended the exploring interpreter idea from [31] in Chapter 2 by adding support for output and different computational effects via a parametric monad. We also captured a set of requirements for language definitions, such that any language satisfying these requirements obtains an exploring interpreter for free, and thus support for exploratory programming. To support the construction of environments for exploratory programming, we constructed an architecture centered around the exploratory programming protocol — a protocol capturing the essence of exploratory programming, inspired by the language server protocol. This protocol solves the $N \times M$ problem, therefore reducing the work required to reuse (N) interfaces for exploratory programming between (M) different language implementations. What this concretely means for our work, is that anybody can design an environment that supports exploratory programming in terms of the protocol, and the environment can then be used with all languages implementing a compliant language server, including *iCoLa*⁺ and the language defined with *iCoLa*⁺.

In Chapter 2 we also looked at the effects of different history mechanisms on the behavior of the underlying data structure keeping track of

the explored programs and configurations. Based on these experiments, we concluded that a tree-based history mechanism is most optimal, since it does not introduce unexplored traces, and the benefits of a graph data structure can still be provided by keeping track of the graph without making the data structure explicit to the user.

Overall, the contributions made in Chapter 2 to exploratory language environments provide a foundational layer to significant parts of this thesis, especially to the contributions made in Chapter 3 and Chapter 4, which are related to both the meta-level and the object-level. Via the contributions made in Chapter 2, languages defined with *iCoLa* and *iCoLa*⁺ automatically obtain support for exploratory programming, and the meta-language *iCoLa*⁺ itself also obtains support for exploratory programming via these contributions. Furthermore, the introduction of the exploratory programming protocol enables independent experimentation with and evaluation of interfaces for exploratory programming, which might be applicable to *iCoLa*⁺ or languages defined using *iCoLa*⁺, but also to languages defined outside of our work.

7.1.2 *Meta-level and Object-level*

To further support exploration of the design space alongside the support for exploratory programming, we introduced a meta-language in Chapter 3 that natively supports the creation of language variants. Ease of variant creation is obtained by an alternative take on abstract syntax that keeps the structure of a language fluid by decoupling operator usage from operator definitions. Additionally, the meta-language supports reuse via language composition. Therefore, languages can be composed to build more complex languages. The alternative take on abstract syntax complements this idea, since existing design choices related to the structure of a language are not fixed and can be modified, increasing opportunities for languages composition.

In Chapter 3 we demonstrated *iCoLa*, an implementation of this meta-language as an EDSL in Haskell, using several advanced functional programming techniques, such as data types à la carte. This worked well to get an initial implementation for experimentation purposes, but was difficult to fully integrate with the concept of exploratory programming, and the implementation requires knowledge of advanced Haskell features when defining languages. To overcome this, we designed and implemented an external DSL in Chapter 4 that also extends the theoretical model from Chapter 3 with support for user-defined concrete syntax and support for an arbitrary amount of semantics domains. These changes made it easier to fully support exploratory programming at the meta-level, and enable language designers to introduce

language-specific syntax, which can positively affect the experience when writing programs.

All in all, the contributions made in Chapter 3 and Chapter 4 provide a meta-level environment that promotes and simplifies variant creation while building on existing language definitions to reduce work. This combines with the support for exploratory programming, creates an environment in which variant creation and variant management is at the forefront. Furthermore, the transition from meta-level to object-level, and back, to experiment with a defined language and re-evaluate design choices based on that experimentation, is possible in the same environment, therefore smoothening such transitions, thus bringing us closer to our vision of exploratory language development.

7.1.3 *Tooling for Free*

The third categorie of contributions was to support program construction with auxiliary tooling such that more substantive programs can be easier constructed using a defined language. A requirement in the context of exploratory language development, is that such tooling should be obtained without much effort from the language designer. Otherwise, every new language variant might require updates to the implementation of tooling to support the newly defined language, possibly demotivating the language designer to create many variants. To that end, we have contributed two approaches that derive tooling purely from the operational semantics of specific types of languages. In particular, we looked at auto-completion services and debuggers for non-deterministic languages.

We focused on auto-completion services for dynamically typed languages, with a specific focus on approaches that aim to achieve sound completion candidates. Soundness in this context means that completion candidates do not introduce bugs, with our focus being on bugs related to name binding. Our approach utilizes abstract interpretation to build an over-approximating scope graph. To support querying such scope graph, we have extended the scope graph framework with annotations and context-dependent name resolution. Annotations ensure that the over-approximation of the name binding can build a collection of identifiers of which it is certain that they are definitely in scope. Context-dependent name resolution can be seen as an optimization by enabling the merging of scope graphs from different execution points while retaining sound queries. Merging also reduces memory usage and opens up possibilities to perform incremental updates to scope graphs when a program is modified.

In Chapter 6, we focused on debuggers for non-deterministic languages, in particular multiverse debuggers. Multiverse debuggers take

a user-controlled approach to the search through the state space of a program. We utilized an existing framework for multiverse debugging to collect requirements for multiverse debuggers via three case studies, one of which was targeting funcons. Using these requirements, we defined a new framework that generalizes the existing framework, primarily on the method for storing the debug history and on how automatic search through the state space is performed. By utilizing funcons as one of our case studies, we obtain a multiverse debugger for funcons. This debugger can also be used by the languages defined in *iCoLa* and *iCoLa*⁺ that use the funcon semantic domain, thus obtaining a debugger for programs at the object-level for free.

7.2 FUTURE WORK

The work presented in this thesis takes several steps towards the vision of exploratory language development, but does not realize it in full. Several aspects are not fully worked out, and many design considerations and implementation details need to be investigated. We utilize the aforementioned three-axis of contributions to present some of the directions that are still required.

7.2.1 *Exploratory Programming Environments*

Our focus in this thesis has been primarily on the derivation of exploring interpreters from sufficient requirements on a language definition, and the effects of different history mechanisms on user interaction. The architecture and protocol provide the means to develop interfaces for exploratory programming and exploratory language development in an isolated and independent manner. However, we have not yet investigated the direction of interfaces for exploratory programming, and the presented interfaces are simple read-eval-print loops with exploratory programming support. This seems far from ideal, and a well-designed environment can significantly improve the user-experience of exploratory language development. An interface can aid management of variants, but also simplify information extraction out of different experiments with different variants, and possibly simplify variant creation itself.

Management of, and supporting the creation of, variants is also something that can still be explored outside of the context of interfaces for exploratory programming. For example, investigating automatic approaches to the creation of branches in an exploratory session, but also the automatic management of the size of the exploration tree. A big exploration tree not only uses memory, it can also overwhelm a user that wants to navigate the tree. An idea into this direction is to

use existing Git repositories to train a machine learning algorithm to identify situations where branching might be beneficial, which can then be suggested to a user in an exploratory session. Whether Git branches provide the right amount of granularity for the suggestions to be fruitful, is something that needs to be investigated.

7.2.2 *Meta-level and Object-level*

The current implementation of $iCoLa^+$ provides a sufficient environment to experiment with the concept of exploratory language development. However, certain features are required to improve the user experience and simplify language definitions. As mentioned in Chapter 4, being able to decompose semantic functions into helper functions or only define semantic functions for a subset of the operators, can significantly help in reducing the size of the translation functions.

Another aspect that can be investigated is the use of different semantic domains in $iCoLa^+$ to define the semantics of language variants. Currently, we have used funcons in our investigations, and demonstrated that it is possible to define new semantic domains in $iCoLa^+$. However, an interesting investigation would be to explore different techniques for the modular definition of programming language semantics, possibly using different semantic styles, such as denotational semantics or axiomatic semantics. The call-by-push-value calculus [119] seems to be an interesting starting point, especially when combined with algebraic effects and handlers [129, 163, 203] to support computational effects in a modular fashion. Such experimentation would utilize $iCoLa^+$ as a tool for the exploration of approaches for modular semantics definitions of programming language, which is something that we have unexplored on purpose in this thesis. For instance, we have not included the semantic engineer in the overview figure (Figure 7.1). To actively support this activity, $iCoLa^+$ might need to satisfy new requirements aligning with the activity, which needs to be investigated in future work.

Finally, the current user base of $iCoLa^+$ is small, maybe even just me. This makes it impossible to demonstrate and classify the effects of exploratory language development as laid out in this thesis on language design. In future work, user experiments with $iCoLa^+$ could be performed to gain a better understanding in this direction. To do this satisfactorily, $iCoLa^+$, and the environment in which this activity is performed, need to be extended and improved, which requires some of the earlier points in this section to be handled first.

7.2.3 Tooling for Free

In this thesis we have explored two directions of tooling. Many more directions remain to be explored, for example, refactoring tools. Another area that can be investigated is the usage of large language models (LLMs) as a tool to aid the activity of program construction of newly defined languages. This requires investigations into integrating unknown (domain-specific) languages into the knowledge of LLMs. The introduction of large language models also places a uncertainty on programming languages and possibly on this work, as their introduction might make programming an activity that is performed in natural language. Nevertheless, we expect that programming language will not lose relevance because the LLMs still generate code. In this context, the development of new languages can aid this process by designing the language with LLMs in mind. For example, a language can be designed with generation by LLMs in mind, or be designed with a focus on readability to make it easier to evaluate the code generated by LLMs. Advantages of DSLs can also be observed in this domain. For example, a more expressive language for a domain might reduce the lines needed for a solution, which can result in smaller context when prompting an LLM where code is included in the prompt.

For the existing tooling, the work presented in Chapter 5 does not yet fully integrate with the rest of the work. This is due to the fact that the funcon rules do not follow the requirement set out in Chapter 5, that requires explicit handling of heaps and frames in the operational rules. A possible direction to solve this, is to automatically derive such rules from the current funcon rules. Alternatively, the current funcon rules can be adjusted to work with the approach by writing an abstract interpreter for funcons that builds the over-approximation of name binding. This would still be useful, since languages defined with funcons then obtain the tooling as well.

Finally, the overview figure (Figure 7.1) does not have a dividing line between the meta-level and object-level on the tooling side, because tooling can be utilized at both levels. This is demonstrated with the debugger framework presented in Chapter 6, where a language engineering can also use the debugger. However, a debugger for $iCoLa^+$ itself does not exist yet. A possible direction to explore, is to define $iCoLa^+$ in terms of $iCoLa^+$, so to bootstrap $iCoLa^+$. This would have the effect that tooling derived from semantics specifications not only derives tooling for object languages, but also for the meta-language itself. How difficult this is to achieve and how it affects the interactions with $iCoLa^+$ needs to be investigated in future work.

7.3 FINAL REMARKS

Exploratory language development describes a vision where new language variants are easy to create, and experimentation with these variants guides design choices, both at the initial design but also at stages where the language is evolving. To make this a reality, a significant amount of effort is still required, both on the engineering side, and on the evaluation side. The engineering side is actively explored, with recent master's students making valuable additions to the work presented in this thesis, such as a more memory efficient approach for exploratory programming [69], and a more performant implementation of funcons [200]. On the evaluation side, it needs to be determined whether the combination of exploratory programming and language development is the right way to achieve the vision. With the work presented in this thesis, a foundation is built that makes it possible to explore this particular avenue towards realizing this vision.

BIBLIOGRAPHY

- [1] Harold Abelson and Gerald J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 1985. ISBN: 0-262-51036-7.
- [2] Ali Afroozeh, Jean-Christophe Bach, Mark van den Brand, Adrian Johnstone, Maarten Manders, Pierre-Etienne Moreau, and Elizabeth Scott. "Island Grammar-Based Parsing Using GLL and Tom." In: *Software Language Engineering, 5th International Conference, SLE 2012, Dresden, Germany, September 26-28, 2012, Revised Selected Papers*. Ed. by Krzysztof Czarnecki and Görel Hedin. Vol. 7745. Lecture Notes in Computer Science. Springer, 2012, pp. 224–243. DOI: 10.1007/978-3-642-36089-3_13.
- [3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley series in computer science / World student series edition. Addison-Wesley, 1986, pp. I–X, 1–796. ISBN: 0201101947.
- [4] Hyacinth Ali, Gunter Mussbacher, and Jörg Kienzle. "Perspectives to promote modularity, reusability, and consistency in multi-language systems." In: *Innov. Syst. Softw. Eng.* 18.1 (2022), pp. 5–37. DOI: 10.1007/s11334-021-00425-3.
- [5] Hendrik van Antwerpen, Pierre Neron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. "A constraint language for static semantic analysis based on scope graphs." In: *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. Ed. by Martin Erwig and Tiark Ropmf. ACM, 2016, pp. 49–60. DOI: 10.1145/2847538.2847543.
- [6] Hendrik van Antwerpen, Casper Bach Poulsen, Arjen Rouvoet, and Eelco Visser. "Scopes as types." In: *Proc. ACM Program. Lang.* 2.OOPSLA (2018), 114:1–114:30. DOI: 10.1145/3276484.

- [7] Sven Apel and Christian Kästner. "An Overview of Feature-Oriented Software Development." In: *J. Object Technol.* 8.5 (2009), pp. 49–84. DOI: 10.5381/jot.2009.8.5.c5.
- [8] Andrew W. Appel and Jens Palsberg. *Modern Compiler Implementation in Java, 2nd edition*. Cambridge University Press, 2002. ISBN: 0-521-82060-X.
- [9] Krzysztof R. Apt. *From logic programming to Prolog*. Prentice Hall International series in computer science. Prentice Hall, 1997. ISBN: 978-0-13-230368-2.
- [10] Joe Armstrong. "A history of Erlang." In: *Proceedings of the Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III), San Diego, California, USA, 9-10 June 2007*. Ed. by Barbara G. Ryder and Brent Hailpern. ACM, 2007, pp. 1–26. DOI: 10.1145/1238844.1238850.
- [11] Egidio Astesiano. "Inductive and Operational Semantics." In: *IFIP State-of-the-Art Reports, Formal Descriptions of Programming Concepts*. Ed. by E.J. Neuhold and M. Paul. Springer, 1991, pp. 51–136.
- [12] Colin Atkinson and Thomas Kühne. "The Essence of Multilevel Metamodeling." In: *UML 2001 - The Unified Modeling Language, Modeling Languages, Concepts, and Tools, 4th International Conference, Toronto, Canada, October 1-5, 2001, Proceedings*. Ed. by Martin Gogolla and Cris Kobryn. Vol. 2185. Lecture Notes in Computer Science. Springer, 2001, pp. 19–33. DOI: 10.1007/3-540-45441-1_3.
- [13] Casper Bach Poulsen and Peter D. Mosses. "Generating Specialized Interpreters for Modular Structural Operational Semantics." In: *23rd International Symposium on Logic-Based Program Synthesis and Transformation*. Springer, 2014, pp. 220–236.
- [14] Patrick Bahr and Tom Hvitved. "Compositional data types." In: *Proceedings of the seventh ACM SIGPLAN workshop on Generic programming, WGP@ICFP 2011, Tokyo, Japan, September 19-21, 2011*. Ed. by Jaakko Järvi and Shin-Cheng Mu. ACM, 2011, pp. 83–94. DOI: 10.1145/2036918.2036930.

- [15] Mikhail Barash. “Vision: the next 700 language workbenches.” In: *SLE '21: 14th ACM SIGPLAN International Conference on Software Language Engineering, Chicago, IL, USA, October 17 - 18, 2021*. Ed. by Eelco Visser, Dimitris S. Kolovos, and Emma Söderberg. ACM, 2021, pp. 16–21. DOI: 10.1145/3486608.3486907.
- [16] John G. P. Barnes. “An Overview of Ada.” In: *Softw. Pract. Exp.* 10.11 (1980), pp. 851–887. DOI: 10.1002/SPE.4380101102.
- [17] Earl T. Barr and Mark Marron. “Tardis: affordable time-travel debugging in managed runtimes.” In: *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*. Ed. by Andrew P. Black and Todd D. Millstein. ACM, 2014, pp. 67–82. DOI: 10.1145/2660193.2660209.
- [18] Hendrikus J. S. Basten and Jurgen J. Vinju. “Parse Forest Diagnostics with Dr. Ambiguity.” In: *Software Language Engineering*. Ed. by Anthony Sloane and Uwe Aßmann. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 283–302. ISBN: 978-3-642-28830-2.
- [19] Birthe van den Berg, Tom Schrijvers, Casper Bach Poulsen, and Nicolas Wu. “Latent Effects for Reusable Language Components.” In: *Programming Languages and Systems*. Ed. by Hakjoo Oh. Cham: Springer International Publishing, 2021, pp. 182–201. DOI: 10.1007/978-3-030-89051-3_11.
- [20] Francesco Bertolotti, Walter Cazzola, and Luca Favalli. “On the granularity of linguistic reuse.” In: *Journal of Systems and Software* 202 (2023), p. 111704. ISSN: 0164-1212. DOI: <https://doi.org/10.1016/j.jss.2023.111704>.
- [21] M. Beth Kery and B. A. Myers. “Exploring exploratory programming.” In: *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 2017, pp. 25–29. DOI: 10.1109/VLHCC.2017.8103446.
- [22] Kurt William Beyer and Cathryn Carson. “Grace hopper and the early history of computer programming, 1944–1960.” AAI3082108. PhD thesis. 2002.

- [23] L. Thomas van Binsbergen. “Funcons for HGMP: the fundamental constructs of homogeneous generative meta-programming (short paper).” In: *Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2018, Boston, MA, USA, November 5-6, 2018*. Ed. by Eric Van Wyk and Tiark Rumpf. ACM, 2018, pp. 168–174. DOI: 10.1145/3278122.3278132.
- [24] L. Thomas van Binsbergen. “Executable Formal Specification of Programming Languages with Reusable Components.” PhD thesis. Royal Holloway, University of London, 2019.
- [25] L. Thomas van Binsbergen. “Executable formal specification of programming languages with reusable components.” PhD thesis. Royal Holloway, University of London, Egham, UK, 2019.
- [26] L. Thomas van Binsbergen, **Damian Frölich**, Mauricio Verrano Merino, Joey Lai, Pierre Jeanjean, Tijs van der Storm, Benoit Combemale, and Olivier Barais. “A Language-Parametric Approach to Exploratory Programming Environments.” In: *Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2022, Auckland, New Zealand, December 6-7, 2022*. Ed. by Bernd Fischer, Lola Burgueño, and Walter Cazzola. ACM, 2022, pp. 175–188. DOI: 10.1145/3567512.3567527.
- [27] L. Thomas van Binsbergen, Lu-Chi Liu, Robert van Doesburg, and Tom van Engers. “eFLINT: A Domain-Specific Language for Executable Norm Specifications.” In: *Proceedings of the 19th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. GPCE 2020. ACM, 2020.
- [28] L. Thomas van Binsbergen, Peter D. Mosses, and Neil Sculthorpe. “Executable Component-Based Semantics.” In: *Journal of Logical and Algebraic Methods in Programming* 103 (Feb. 2019), pp. 184–212. DOI: 10.1016/j.jlamp.2018.12.004.

- [29] L. Thomas van Binsbergen, Peter D. Mosses, and Neil Sculthorpe. “Executable component-based semantics.” In: *J. Log. Algebraic Methods Program.* 103 (2019), pp. 184–212. DOI: 10.1016/j.jlamp.2018.12.004.
- [30] L. Thomas van Binsbergen, Elizabeth Scott, and Adrian Johnstone. “Purely functional GLL parsing.” In: *J. Comput. Lang.* 58 (2020), p. 100945. DOI: 10.1016/j.cola.2020.100945.
- [31] L. Thomas van Binsbergen, Mauricio Verano Merino, Pierre Jeanjean, Tijs van der Storm, Benoit Combemale, and Olivier Barais. “A Principled Approach to REPL Interpreters.” In: *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software.* ACM, 2020, pp. 84–100. DOI: 10.1145/3426428.3426917.
- [32] Erwan Bousse, Dorian Leroy, Benoit Combemale, Manuel Wimmer, and Benoit Baudry. “Omniscient debugging for executable DSLs.” In: *Journal of Systems and Software* 137 (2018), pp. 261–288.
- [33] Erwan Bousse, Dorian Leroy, Benoit Combemale, Manuel Wimmer, and Benoit Baudry. “Omniscient debugging for executable DSLs.” In: *J. Syst. Softw.* 137 (2018), pp. 261–288. DOI: 10.1016/J.JSS.2017.11.025.
- [34] Jonathan Immanuel Brachthäuser and Philipp Schuster. “Effekt: extensible algebraic effects in Scala (short paper).” In: *Proceedings of the 8th ACM SIGPLAN International Symposium on Scala.* SCALA 2017. Vancouver, BC, Canada: Association for Computing Machinery, 2017, pp. 67–72. ISBN: 9781450355292. DOI: 10.1145/3136000.3136007.
- [35] Edwin C. Brady. “Programming and reasoning with algebraic effects and dependent types.” In: *ACM SIGPLAN International Conference on Functional Programming, ICFP’13, Boston, MA, USA - September 25 - 27, 2013.* Ed. by Greg Morrisett and Tarmo Uustalu. ACM, 2013, pp. 133–144. DOI: 10.1145/2500365.2500581.

- [36] Arvid Butting, Jerome Pfeiffer, Bernhard Rumpe, and Andreas Wortmann. "A compositional framework for systematic modeling language reuse." In: *MoDELS '20: ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems, Virtual Event, Canada, 18-23 October, 2020*. Ed. by Eugene Syriani, Houari A. Sahraoui, Juan de Lara, and Silvia Abrahão. ACM, 2020, pp. 35–46. DOI: 10.1145/3365438.3410934.
- [37] Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. "Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages." In: *J. Funct. Program.* 19.5 (2009), pp. 509–543. DOI: 10.1017/S0956796809007205.
- [38] Walter Cazzola. "Domain-Specific Languages in Few Steps - The Neverlang Approach." In: *Software Composition - 11th International Conference, SC@TOOLS 2012, Prague, Czech Republic, May 31 - June 1, 2012. Proceedings*. Ed. by Thomas Gschwind, Flavio De Paoli, Volker Gruhn, and Matthias Book. Vol. 7306. Lecture Notes in Computer Science. Springer, 2012, pp. 162–177. DOI: 10.1007/978-3-642-30564-1_11.
- [39] Walter Cazzola and Diego Mathias Olivares. "Gradually Learning Programming Supported by a Growable Programming Language." In: *IEEE Trans. Emerg. Top. Comput.* 4.3 (2016), pp. 404–415. DOI: 10.1109/TETC.2015.2446192.
- [40] N. Chomsky and M.P. Schützenberger. "The Algebraic Theory of Context-Free Languages*." In: *Computer Programming and Formal Systems*. Ed. by P. Braffort and D. Hirschberg. Vol. 35. Studies in Logic and the Foundations of Mathematics. Elsevier, 1963, pp. 118–161. DOI: [https://doi.org/10.1016/S0049-237X\(08\)72023-8](https://doi.org/10.1016/S0049-237X(08)72023-8).
- [41] Martin Churchill, Peter D. Mosses, Neil Sculthorpe, and Paolo Torrini. "Reusable Components of Semantic Specifications." In: *Transactions on Aspect-Oriented Software Development XII. TAOSD 2015*. 2015, pp. 132–179.
- [42] Matteo Cimini. "Languages as first-class citizens (vision paper)." In: *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2018, Boston, MA, USA, November 05-06, 2018*. Ed. by David

- J. Pearce, Tanja Mayerhofer, and Friedrich Steimann. ACM, 2018, pp. 65–69. DOI: 10.1145/3276604.3276983.
- [43] Paul Clements and Linda Northrop. *Software product lines*. Addison-Wesley Boston, 2002.
- [44] Benoit Combemale, Jörg Kienzle, et al. “Concern-oriented language development (COLD): Fostering reuse in language engineering.” In: *Comput. Lang. Syst. Struct.* 54 (2018), pp. 139–155. DOI: 10.1016/j.cl.2018.05.004.
- [45] LUKAS CONVENT, SAM LINDLEY, CONOR MCBRIDE, and CRAIG MCLAUGHLIN. “Doo bee doo bee doo.” In: *Journal of Functional Programming* 30 (2020), e9. DOI: 10.1017/S0956796820000039.
- [46] Mischa Corsius, Stijn Hoppenbrouwers, Mariette Lokin, Elian Baars, Gertrude Sangers-Van Cappellen, and Ilona Wilmont. “RegelSpraaK: A CNL for executable tax rules specification.” In: *Proceedings of the seventh international workshop on controlled natural language (CNL 2020/21)*. 2021.
- [47] Patrick Cousot. *Principles of abstract interpretation*. MIT Press, 2021.
- [48] Patrick Cousot and Radhia Cousot. “Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints.” In: *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*. Ed. by Robert M. Graham, Michael A. Harrison, and Ravi Sethi. ACM, 1977, pp. 238–252. DOI: 10.1145/512950.512973.
- [49] Douglas A. Creager and Hendrik van Antwerpen. “Stack Graphs: Name Resolution at Scale.” In: *Eelco Visser Commemorative Symposium, EVCS 2023, April 5, 2023, Delft, The Netherlands*. Ed. by Ralf Lämmel, Peter D. Mosses, and Friedrich Steimann. Vol. 109. OASICS. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, 8:1–8:12. DOI: 10.4230/OASICS.EVCS.2023.8.

- [50] Daniela S Cruzes and Lotfi ben Othmane. “Threats to validity in empirical software security research.” In: *Empirical research for software security*. CRC Press, 2017, pp. 275–300.
- [51] **Damian Frölich** and L. Thomas van Binsbergen. “A Generic Back-End for Exploratory Programming.” In: *Trends in Functional Programming - 22nd International Symposium, TFP 2021, Virtual Event, February 17-19, 2021, Revised Selected Papers*. Ed. by Viktória Zsóck and John Hughes. Vol. 12834. Lecture Notes in Computer Science. Springer, 2021, pp. 24–43. DOI: 10.1007/978-3-030-83978-9_2.
- [52] **Damian Frölich** and L. Thomas van Binsbergen. “iCoLa: A Compositional Meta-language with Support for Incremental Language Development.” In: *Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2022, Auckland, New Zealand, December 6-7, 2022*. Ed. by Bernd Fischer, Lola Burgueño, and Walter Cazzola. ACM, 2022, pp. 202–215. DOI: 10.1145/3567512.3567529.
- [53] **Damian Frölich** and L. Thomas van Binsbergen. “On the Soundness of Auto-completion Services for Dynamically Typed Languages.” In: *Proceedings of the 23rd ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences. GPCE ’24*. Pasadena, CA, USA: Association for Computing Machinery, 2024, pp. 107–120. ISBN: 9798400712111. DOI: 10.1145/3689484.3690734.
- [54] **Damian Frölich**, Tommaso Pacciani, and L. Thomas van Binsbergen. “Exploratory, Omniscient, and Multiverse Diagnostics in Debuggers for Non-Deterministic Languages.” In: *Proceedings of the 18th ACM SIGPLAN International Conference on Software Language Engineering. SLE ’25*. Koblenz, Germany: Association for Computing Machinery, 2025, pp. 134–147. ISBN: 9798400718847. DOI: 10.1145/3732771.3742719.
- [55] **Damian Frölich** and L. Thomas van Binsbergen. “iCoLa+: An extensible meta-language with support for exploratory language development.” In: *Journal of Systems and Software*

- 211 (2024), p. 111979. ISSN: 0164-1212. DOI: <https://doi.org/10.1016/j.jss.2024.111979>.
- [56] Nikita Danilenko. “Designing Functional Implementations of Graph Algorithms (Entwurf funktionaler Implementierungen von Graphalgorithmen).” PhD thesis. Kiel University, Germany, 2015. URL: <https://nbn-resolving.org/urn:nbn:de:gbv:8-diss-186649>.
- [57] David Darais, Nicholas Labich, Phuc C. Nguyen, and David Van Horn. “Abstracting definitional interpreters (functional pearl).” In: *Proc. ACM Program. Lang.* 1.ICFP (2017), 12:1–12:25. DOI: 10.1145/3110256.
- [58] Donald W. Davies. “THE BOMBE A REMARKABLE LOGIC MACHINE.” In: *Cryptologia* 23.2 (1999), pp. 108–138. DOI: 10.1080/0161-119991887793.
- [59] Thomas Dague, Benoit Combemale, Arnaud Blouin, Olivier Barais, and Jean-Marc Jézéquel. “Mélange: a meta-language for modular and reusable development of DSLs.” In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering, SLE 2015, Pittsburgh, PA, USA, October 25-27, 2015*. Ed. by Richard F. Paige, Davide Di Ruscio, and Markus Völter. ACM, 2015, pp. 25–36. DOI: 10.1145/2814251.2814252.
- [60] Edsger W. Dijkstra. “Guarded Commands, Nondeterminacy and Formal Derivation of Programs.” In: *Commun. ACM* 18.8 (1975), pp. 453–457. DOI: 10.1145/360933.360975.
- [61] Robert van Doesburg and Tom M. van Engers. “Perspectives on the Formal Representation of the Interpretation of Norms.” In: *Legal Knowledge and Information Systems - JURIX 2016: The Twenty-Ninth Annual Conference*. Ed. by Floris Bex and Serena Villata. Vol. 294. Frontiers in Artificial Intelligence and Applications. IOS Press, 2016, pp. 183–186. DOI: 10.3233/978-1-61499-726-9-183.
- [62] Jay Earley. “An Efficient Context-Free Parsing Algorithm.” In: *Commun. ACM* 13.2 (1970), pp. 94–102. DOI: 10.1145/362007.362035.

- [63] Torbjörn Ekman and Görel Hedin. “The jastadd extensible java compiler.” In: *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*. Ed. by Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr. ACM, 2007, pp. 1–18. DOI: 10.1145/1297027.1297029.
- [64] John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen C North, and Gordon Woodhull. “Graphviz—open source graph drawing tools.” In: *Graph Drawing: 9th International Symposium, GD 2001 Vienna, Austria, September 23–26, 2001 Revised Papers 9*. Springer, 2002, pp. 483–484.
- [65] Sebastian Erdweg, Paolo G. Giarrusso, and Tillmann Rendel. “Language Composition Untangled.” In: *Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications. LDTA '12*. Tallinn, Estonia: ACM, 2012. ISBN: 9781450315364. DOI: 10.1145/2427048.2427055.
- [66] Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. “SugarJ: library-based syntactic language extensibility.” In: *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*. Ed. by Cristina Videira Lopes and Kathleen Fisher. ACM, 2011, pp. 391–406. DOI: 10.1145/2048066.2048099.
- [67] Sebastian Erdweg et al. “The State of the Art in Language Workbenches - Conclusions from the Language Workbench Challenge.” In: *Software Language Engineering - 6th International Conference, SLE 2013, Indianapolis, IN, USA, October 26-28, 2013. Proceedings*. Ed. by Martin Erwig, Richard F. Paige, and Eric Van Wyk. Vol. 8225. Lecture Notes in Computer Science. Springer, 2013, pp. 197–217. DOI: 10.1007/978-3-319-02654-1_11.
- [68] Sebastian Erdweg et al. “Evaluating and comparing language workbenches: Existing results and benchmarks for the future.” In: *Computer Languages, Systems & Structures* 44 (2015), pp. 24–47. ISSN: 1477-8424.

- [69] Olaf Erkemeij. “Exploring Efficient Storage of State in Exploratory Programming.” MA thesis. University of Amsterdam, 2025.
- [70] Robert Fenichel and Jerome C. Yochelson. “A LISP garbage-collector for virtual-memory computer systems.” In: *Commun. ACM* 12.11 (1969), pp. 611–612. DOI: 10.1145/363269.363280.
- [71] Maarten Fokkinga, Johan Jeuring, Lambert Meertens, and Erik Meijer. “A translation from attribute grammars to catamorphisms.” In: *Squiggolist* 2.1 (1991), pp. 20–26.
- [72] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. “Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem.” In: *ACM Trans. Program. Lang. Syst.* 29.3 (2007), p. 17. DOI: 10.1145/1232420.1232424.
- [73] Damian Frolich and Thomas van Binsbergen. *A selection of Python programs with key name binding challenges for auto-completion services*. Sept. 2024. DOI: 10.5281/zenodo.13628718.
- [74] Richard A. Frost, Rahmatullah Hafiz, and Paul Callaghan. “Parser Combinators for Ambiguous Left-Recursive Grammars.” In: *Practical Aspects of Declarative Languages*. Vol. 4902. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2008, pp. 167–181. ISBN: 978-3-540-77441-9. DOI: 10.1007/978-3-540-77442-6_12.
- [75] J. A. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright. “Initial Algebra Semantics and Continuous Algebras.” In: *Journal of the ACM* 24.1 (Jan. 1977), pp. 68–95. ISSN: 0004-5411.
- [76] J. A. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright. “Initial Algebra Semantics and Continuous Algebras.” In: *Journal of the ACM* 24.1 (1977), pp. 68–95. ISSN: 0004-5411. DOI: 10.1145/321992.321997.

- [77] M. Gordon, R. Milner, L. Morris, M. Newey, and C. Wadsworth. "A Metalanguage for interactive proof in LCF." In: *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '78. Tucson, Arizona: Association for Computing Machinery, 1978, pp. 119–130. ISBN: 9781450373487. DOI: 10.1145/512760.512773.
- [78] Maria Gouseti, Chiel Peters, and Tijs van der Storm. "Extensible language implementation with object algebras (short paper)." In: *Generative Programming: Concepts and Experiences, GPCE'14, Vasteras, Sweden, September 15-16, 2014*. Ed. by Ulrik Pagh Schultz and Matthew Flatt. ACM, 2014, pp. 25–28. DOI: 10.1145/2658761.2658765.
- [79] Dick Grune. *Parsing Techniques: A Practical Guide*. 2nd. Springer Publishing Company, Incorporated, 2010. ISBN: 9781441919014.
- [80] Esther Guerra, Juan de Lara, Marsha Chechik, and Rick Salay. "Property Satisfiability Analysis for Product Lines of Modelling Languages." In: *IEEE Trans. Software Eng.* 48.2 (2022), pp. 397–416. DOI: 10.1109/TSE.2020.2989506.
- [81] Thomas Haigh and Paul E Ceruzzi. *A new history of modern computing*. MIT Press, 2021.
- [82] Brian Hayes. "Thoughts on Mathematica." In: *Pixel* 1. January/February (1990), pp. 28–34.
- [83] Jan Heering, P. R. H. Hendriks, Paul Klint, and J. Rekers. "The syntax definition formalism SDF - reference manual." In: *ACM SIGPLAN Notices* 24.11 (1989), pp. 43–75. DOI: 10.1145/71605.71607.
- [84] Felienne Hermans. "Hedy: A Gradual Language for Programming Education." In: *ICER 2020: International Computing Education Research Conference, Virtual Event, New Zealand, August 10-12, 2020*. Ed. by Anthony V. Robins, Adon Moskal, Amy J. Ko, and Renée McCauley. ACM, 2020, pp. 259–270. DOI: 10.1145/3372782.3406262.
- [85] Paul Hudak. "Building Domain-Specific Embedded Languages." In: *ACM Comput. Surv.* 28.4es (1996), p. 196. DOI: 10.1145/242224.242477.

- [86] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. "A History of Haskell: Being Lazy with Class." In: *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*. HOPL III. San Diego, California: ACM, 2007. ISBN: 978-1-59593-766-7. DOI: 10.1145/1238844.1238856.
- [87] Pablo Inostroza and Tijs van der Storm. "Modular interpreters with implicit context propagation." In: *Comput. Lang. Syst. Struct.* 48 (2017), pp. 39–67. DOI: 10.1016/j.cl.2016.08.001.
- [88] Saraiva Joao. *Purely functional implementation of attribute grammars*. 1999.
- [89] Patricia Johann and Neil Ghani. "Foundations for structured programming with GADTs." In: *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*. Ed. by George C. Necula and Philip Wadler. ACM, 2008, pp. 297–308. DOI: 10.1145/1328438.1328475.
- [90] Gwendal Jouneaux, **Damian Frölich**, Olivier Barais, Benoit Combemale, Gurvan Le Guernic, Gunter Mussbacher, and L. Thomas van Binsbergen. "Adaptive Structural Operational Semantics." In: *Proceedings of the 16th ACM SIGPLAN International Conference on Software Language Engineering*. SLE 2023. Cascais, Portugal: Association for Computing Machinery, 2023, pp. 29–42. ISBN: 9798400703966. DOI: 10.1145/3623476.3623517.
- [91] Gilles Kahn. "Natural Semantics." In: *Proceedings of the 4th Annual Symposium on Theoretical Aspects of Computer Science*. Springer-Verlag, 1987, pp. 22–39.
- [92] Gilles Kahn. "Natural Semantics." In: *STACS 87, 4th Annual Symposium on Theoretical Aspects of Computer Science, Passau, Germany, February 19-21, 1987, Proceedings*. Ed. by Franz-Josef Brandenburg, Guy Vidal-Naquet, and Martin Wirsing. Vol. 247. Lecture Notes in Computer Science. Springer, 1987, pp. 22–39. DOI: 10.1007/BFB0039592.

- [93] John B. Kam and Jeffrey D. Ullman. "Monotone Data Flow Analysis Frameworks." In: *Acta Informatica* 7 (1977), pp. 305–317. DOI: 10.1007/BF00290339.
- [94] Lennart C. L. Kats and Eelco Visser. "The Spoofox language workbench." In: *Companion to the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, part of SPLASH 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*. Ed. by William R. Cook, Siobhán Clarke, and Martin C. Rinard. ACM, 2010, pp. 237–238. DOI: 10.1145/1869542.1869592.
- [95] Lennart C. L. Kats and Eelco Visser. "The Spoofox Language Workbench: Rules for Declarative Specification of Languages and IDEs." In: *International Conference on Object Oriented Programming Systems Languages and Applications. OOPSLA 2010*. ACM, 2010, pp. 444–463. DOI: 10.1145/1869459.1869497.
- [96] Lennart C. L. Kats and Eelco Visser. "The spoofox language workbench: rules for declarative specification of languages and IDEs." In: *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*. Ed. by William R. Cook, Siobhán Clarke, and Martin C. Rinard. ACM, 2010, pp. 444–463. DOI: 10.1145/1869459.1869497.
- [97] Sven Keidel, Casper Bach Poulsen, and Sebastian Erdweg. "Compositional soundness proofs of abstract interpreters." In: *Proc. ACM Program. Lang.* 2.ICFP (2018), 72:1–72:26. DOI: 10.1145/3236767.
- [98] Ken Kennedy. "Use-Definition Chains with Applications." In: *Comput. Lang.* 3.3 (1978), pp. 163–179. DOI: 10.1016/0096-0551(78)90009-7.
- [99] Mary Beth Kery and Brad A. Myers. "Exploring exploratory programming." In: *2017 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2017, Raleigh, NC, USA, October 11-14, 2017*. Ed. by Austin Z. Henley, Peter Rogers, and Anita Sarma. IEEE Computer Society, 2017, pp. 25–29. DOI: 10.1109/VLHCC.2017.8103446.

- [100] Uday P. Khedker, Amitabha Sanyal, and Bageshri Sathe. *Data Flow Analysis - Theory and Practice*. CRC Press, 2009. ISBN: 978-0-8493-2880-0.
- [101] Donnacha Oisín Kidney and Nicolas Wu. “Formalising Graph Algorithms with Coinduction.” In: *Proc. ACM Program. Lang.* 9.POPL (2025), pp. 1657–1686. DOI: 10.1145/3704892.
- [102] Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. “Code Prediction by Feeding Trees to Transformers.” In: *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 2021, pp. 150–162. DOI: 10.1109/ICSE43902.2021.00026.
- [103] Oleg Kiselyov and KC Sivaramakrishnan. “Eff Directly in OCaml.” In: *Electronic Proceedings in Theoretical Computer Science* 285 (Dec. 2018), pp. 23–58. ISSN: 2075-2180. DOI: 10.4204/eptcs.285.2.
- [104] Paul Klint. “A Meta-Environment for Generating Programming Environments.” In: *ACM Trans. Softw. Eng. Methodol.* 2.2 (1993), pp. 176–201. DOI: 10.1145/151257.151260.
- [105] Paul Klint, Tijs van der Storm, and Jurgen Vinju. “Rascal: A Domain Specific Language for Source Code Analysis and Manipulation.” In: *Proceedings of the 2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*. IEEE Computer Society, 2009, pp. 168–177. ISBN: 978-0-7695-3793-1. DOI: 10.1109/SCAM.2009.28.
- [106] Thomas Kluyver et al. “Jupyter Notebooks - a publishing format for reproducible computational workflows.” In: *Positioning and Power in Academic Publishing: Players, Agents and Agendas*. Ed. by Fernando Loizides and Birgit Schmidt. Netherlands: IOS Press, 2016, pp. 87–90. DOI: 10.3233/978-1-61499-649-1-87.
- [107] Thomas Kluyver et al. “Jupyter Notebooks - a publishing format for reproducible computational workflows.” In: *Positioning and Power in Academic Publishing: Players, Agents and Agendas, 20th International Conference on Electronic Publishing, Göttingen, Germany, June 7-9, 2016*. Ed. by

- Fernando Loizides and Birgit Schmidt. IOS Press, 2016, pp. 87–90. DOI: 10.3233/978-1-61499-649-1-87.
- [108] Donald E Knuth and Luis Trabb Pardo. “The early development of programming languages.” In: *A history of computing in the twentieth century* (1980), pp. 197–273.
- [109] Robert Kowalski and Keith L. Clark. “Logic programming.” In: *Encyclopedia of Computer Science*. GBR: John Wiley and Sons Ltd., 2003, pp. 1017–1031. ISBN: 0470864125.
- [110] Thomas Kühn, Walter Cazzola, and Diego Mathias Olivares. “Choosy and picky: configuration of language product lines.” In: *Proceedings of the 19th International Conference on Software Product Line, SPLC 2015, Nashville, TN, USA, July 20-24, 2015*. Ed. by Douglas C. Schmidt. ACM, 2015, pp. 71–80. DOI: 10.1145/2791060.2791092.
- [111] Joey Lai. “Supporting Exploratory Programming in Computational Notebooks with an Exploring Interpreter (Working title).” MA thesis. the Netherlands: Universiteit van Amsterdam, 2021.
- [112] Juan de Lara and Esther Guerra. “Deep Meta-modelling with MetaDepth.” In: *Objects, Models, Components, Patterns, 48th International Conference, TOOLS 2010, Málaga, Spain, June 28 - July 2, 2010. Proceedings*. Ed. by Jan Vitek. Vol. 6141. Lecture Notes in Computer Science. Springer, 2010, pp. 1–20. DOI: 10.1007/978-3-642-13953-6_1.
- [113] Juan de Lara and Esther Guerra. “Multi-level Model Product Lines - Open and Closed Variability for Modelling Language Families.” In: *Fundamental Approaches to Software Engineering - 23rd International Conference, FASE 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings*. Ed. by Heike Wehrheim and Jordi Cabot. Vol. 12076. Lecture Notes in Computer Science. Springer, 2020, pp. 161–181. DOI: 10.1007/978-3-030-45234-6_8.
- [114] Juan de Lara and Esther Guerra. “Language Family Engineering with Product Lines of Multi-level Models.” In: *Formal Aspects Comput.* 33.6 (2021), pp. 1173–1208. DOI: 10.1007/s00165-021-00554-3.

- [115] Juan de Lara, Esther Guerra, and Paolo Bottoni. “Modular language product lines: a graph transformation approach.” In: *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems, MODELS 2022, Montreal, Quebec, Canada, October 23-28, 2022*. Ed. by Eugene Syriani, Houari A. Sahraoui, Nelly Bencomo, and Manuel Wimmer. ACM, 2022, pp. 334–344. DOI: 10.1145/3550355.3552444.
- [116] Juan de Lara, Esther Guerra, and Jesús Sánchez Cuadrado. “Model-driven engineering with domain-specific meta-modelling languages.” In: *Softw. Syst. Model.* 14.1 (2015), pp. 429–459. DOI: 10.1007/s10270-013-0367-z.
- [117] David Lazar, Andrei Arusoaie, Traian Florin Şerbănuţă, Chucky Ellison, Radu Mereuta, Dorel Lucanu, and Grigore Roşu. “Executing Formal Semantics with the \mathbb{K} Tool.” In: *International Symposium on Formal Methods*. Vol. 7436. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 267–271. ISBN: 978-3-642-32759-9.
- [118] Daan Leijen. “Koka: Programming with Row Polymorphic Effect Types.” In: *Electronic Proceedings in Theoretical Computer Science* 153 (June 2014), pp. 100–126. ISSN: 2075-2180. DOI: 10.4204/eptcs.153.8.
- [119] Paul Blain Levy. “Call-by-push-value.” PhD thesis. 2013.
- [120] Bil Lewis. “Debugging Backwards in Time.” In: *Computing Research Repository* cs.SE/0310016 (2003). URL: <http://arxiv.org/abs/cs/0310016>.
- [121] Bil Lewis. “Debugging Backwards in Time.” In: *CoRR* cs.SE/0310016 (2003). URL: <http://arxiv.org/abs/cs/0310016>.
- [122] Sheng Liang, Paul Hudak, and Mark Jones. “Monad Transformers and Modular Interpreters.” In: *22nd Symposium on Principles of Programming Languages*. ACM, 1995, pp. 333–343. DOI: 10.1145/199448.199528.
- [123] Jörg Liebig, Rolf Daniel, and Sven Apel. “Feature-oriented language families: A case study.” In: *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems*. 2013, pp. 1–8.

- [124] Adrian Lienhard, Tudor Girba, and Oscar Nierstrasz. "Practical object-oriented back-in-time debugging." In: *European Conference on Object-Oriented Programming*. Springer, 2008, pp. 592–615.
- [125] Carmen Torres Lopez, Robbert Gurdeep Singh, Stefan Marr, Elisa Gonzalez Boix, and Christophe Scholliers. "Multiverse Debugging: Non-Deterministic Debugging for Non-Deterministic Programs (Brave New Idea Paper)." In: *33rd European Conference on Object-Oriented Programming, ECOOP 2019, July 15-19, 2019, London, United Kingdom*. Ed. by Alastair F. Donaldson. Vol. 134. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 27:1–27:30. DOI: 10.4230/LIPICS.ECOOP.2019.27.
- [126] Simon Marlow. *Haskell 2010 Language Report*. 2010.
- [127] Nicholas D. Matsakis and Felix S. Klock II. "The rust language." In: *Proceedings of the 2014 ACM SIGAda annual conference on High integrity language technology, HILT 2014, Portland, Oregon, USA, October 18-21, 2014*. Ed. by Michael B. Feldman and S. Tucker Taft. ACM, 2014, pp. 103–104. DOI: 10.1145/2663171.2663188.
- [128] John McCarthy. "History of LISP." In: *History of Programming Languages, from the ACM SIGPLAN History of Programming Languages Conference, June 1-3, 1978, Los Angeles, California, USA*. Ed. by Richard L. Wexelblat. Academic Press / ACM, 1978, pp. 173–185. DOI: 10.1145/800025.1198360.
- [129] Dylan McDermott. "Grading Call-By-Push-Value, Explicitly and Implicitly." In: *10th International Conference on Formal Structures for Computation and Deduction, FSCD 2025, July 14-20, 2025, Birmingham, UK*. Ed. by Maribel Fernández. Vol. 337. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2025, 28:1–28:19. DOI: 10.4230/LIPICS.FSCD.2025.28.
- [130] Erik Meijer, Maarten M. Fokkinga, and Ross Paterson. "Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire." In: *Functional Programming Languages and Computer Architecture, 5th ACM Conference, Cambridge, MA, USA, August 26-30, 1991, Proceedings*. Ed. by John

- Hughes. Vol. 523. Lecture Notes in Computer Science. Springer, 1991, pp. 124–144. DOI: 10.1007/3540543961_7.
- [131] David Méndez-Acuña, José Angel Galindo, Thomas Degueule, Benoit Combemale, and Benoit Baudry. “Leveraging Software Product Lines Engineering in the development of external DSLs: A systematic literature review.” In: *Comput. Lang. Syst. Struct.* 46 (2016), pp. 206–235. DOI: 10.1016/j.cl.2016.09.004.
- [132] Mauricio Verano Merino, L. Thomas van Binsbergen, and Mazyar Seraj. “Making the Invisible Visible in Computational Notebooks.” In: *2022 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 2022, pp. 1–3. DOI: 10.1109/VL/HCC53370.2022.9833148.
- [133] Marjan Mernik, Jan Heering, and Anthony M. Sloane. “When and how to develop domain-specific languages.” In: *ACM Comput. Surv.* 37.4 (2005), pp. 316–344. DOI: 10.1145/1118890.1118892.
- [134] Marjan Mernik, Mitja Lenic, Enis Avdicausevic, and Viljem Zumer. “Multiple Attribute Grammar Inheritance.” In: *Informatica (Slovenia)* 24.3 (2000).
- [135] Marjan Mernik, Mitja Lenic, Enis Avdicausevic, and Viljem Zumer. “LISA: An Interactive Environment for Programming Language Development.” In: *Compiler Construction, 11th International Conference, CC 2002, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*. Ed. by R. Nigel Horspool. Vol. 2304. Lecture Notes in Computer Science. Springer, 2002, pp. 1–4. DOI: 10.1007/3-540-45937-5_1.
- [136] Marjan Mernik and Viljem Zumer. “Incremental programming language development.” In: *Comput. Lang. Syst. Struct.* 31.1 (2005), pp. 1–16. DOI: 10.1016/j.cl.2004.02.001.
- [137] Luka Miljak, Casper Bach Poulsen, and Flip van Spaendonck. “Verifying Well-Typedness Preservation of Refactorings using Scope Graphs.” In: *Proceedings of the 25th ACM International Workshop on Formal Techniques for Java-*

- like Programs, FTfJP 2023, Seattle, WA, USA, 18 July 2023.* Ed. by Aaron Tomb. ACM, 2023, pp. 44–50. DOI: 10.1145/3605156.3606455.
- [138] Robin Milner, Mads Tofte, and David MacQueen. *The Definition of Standard ML*. MIT Press, 1997. ISBN: 0262631814.
- [139] Eugenio Moggi. “Notions of Computation and Monads.” In: *Information and Computation* 93.1 (1991), pp. 55–92. DOI: 10.1016/0890-5401(91)90052-4.
- [140] Eugenio Moggi. “Notions of Computation and Monads.” In: *Inf. Comput.* 93.1 (1991), pp. 55–92. DOI: 10.1016/0890-5401(91)90052-4.
- [141] Raphaël Monat, Abdelraouf Ouadjaout, and Antoine Miné. “Static Type Analysis by Abstract Interpretation of Python Programs.” In: *34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15–17, 2020, Berlin, Germany (Virtual Conference)*. Ed. by Robert Hirschfeld and Tobias Pape. Vol. 166. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, 17:1–17:29. DOI: 10.4230/LIPICS.ECOOP.2020.17.
- [142] Peter D. Mosses. “Denotational Semantics.” In: *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*. Ed. by Jan van Leeuwen. Elsevier and MIT Press, 1990, pp. 575–631. DOI: 10.1016/b978-0-444-88074-1.50016-0.
- [143] Peter D. Mosses. “Modular Structural Operational Semantics.” In: *Journal of Logic and Algebraic Programming* 60–61 (2004), pp. 195–228.
- [144] Peter D. Mosses. “Software meta-language engineering and CBS.” In: *Journal of Computer Languages* 50 (2019), pp. 39–48. ISSN: 2590-1184. DOI: 10.1016/j.jvlc.2018.11.003.
- [145] Peter D. Mosses. “Fundamental Constructs in Programming Languages.” In: *Leveraging Applications of Formal Methods, Verification and Validation - 10th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2021, Rhodes, Greece, October 17–29, 2021, Proceedings*. Ed.

- by Tiziana Margaria and Bernhard Steffen. Vol. 13036. *Lecture Notes in Computer Science*. Springer, 2021, pp. 296–321. DOI: 10.1007/978-3-030-89159-6_19.
- [146] Peter D. Mosses and Mark J. New. “Implicit Propagation in Structural Operational Semantics.” In: *Electronic Notes in Theoretical Computer Science* 229.4 (2009).
- [147] Peter D. Mosses, Neil Sculthorpe, and L. Thomas Van Binsbergen. *Funcons-Beta*. Online GitHub repository. 2021. URL: <https://plancomps.github.io/CBS-beta/Funcons-beta/>.
- [148] Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. “The Lean Theorem Prover (System Description).” In: *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*. Ed. by Amy P. Felty and Aart Middeldorp. Vol. 9195. *Lecture Notes in Computer Science*. Springer, 2015, pp. 378–388. DOI: 10.1007/978-3-319-21401-6_26.
- [149] Pierre Neron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. “A Theory of Name Resolution.” In: *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*. Ed. by Jan Vitek. Vol. 9032. *Lecture Notes in Computer Science*. Springer, 2015, pp. 205–231. DOI: 10.1007/978-3-662-46669-8_9.
- [150] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of program analysis*. Springer, 1999. ISBN: 978-3-540-65410-0. DOI: 10.1007/978-3-662-03811-6.
- [151] Ulf Norell. *Towards a practical programming language based on dependent type theory*. Vol. 32. Chalmers University of Technology, 2007.
- [152] Pablo Noriega, Amit K. Chopra, Nicoletta Fornara, Henrique Lopes Cardoso, and Munindar P. Singh. “Regulated MAS: Social Perspective.” In: *Normative Multi-Agent Systems*. Ed. by Giulia Andrighetto, Guido Governatori, Pablo Noriega, and Leendert W. N. van der Torre. Vol. 4.

- Dagstuhl Follow-Ups. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2013, pp. 93–133. DOI: 10.4230/DFU.VOL4.12111.93.
- [153] Bruno C. d. S. Oliveira and William R. Cook. “Extensibility for the Masses - Practical Extensibility with Object Algebras.” In: *ECOOP 2012 – Object-Oriented Programming*. Springer Berlin Heidelberg, 2012, pp. 2–27. DOI: 10.1007/978-3-642-31057-7_2.
- [154] Dominic Orchard and Tomas Petricek. “Embedding effect systems in Haskell.” In: *SIGPLAN Not.* 49.12 (Sept. 2014), pp. 13–24. ISSN: 0362-1340. DOI: 10.1145/2775050.2633368.
- [155] Tommaso Pacciani, **Damian Frölich**, L. Thomas van Binsbergen, and Chrysa Papagianni. “P4DDG: Data-Dependent Grammars for Packet Specification and Parsing in P4.” In: *Proceedings of the 24th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. GPCE ’25. Bergen, Norway: Association for Computing Machinery, 2025, pp. 54–66. ISBN: 9798400719950. DOI: 10.1145/3742876.3742879.
- [156] Matthias Pasquier, Ciprian Teodorov, Frédéric Jouault, Matthias Brun, Luka Le Roux, and Loïc Lagadec. “Practical multiverse debugging through user-defined reductions: application to UML models.” In: *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems, MODELS 2022, Montreal, Quebec, Canada, October 23-28, 2022*. Ed. by Eugene Syriani, Houari A. Sahraoui, Nelly Bencomo, and Manuel Wimmer. ACM, 2022, pp. 87–97. DOI: 10.1145/3550355.3552447.
- [157] Matthias Pasquier, Ciprian Teodorov, Frédéric Jouault, Matthias Brun, Luka Le Roux, and Loïc Lagadec. “Temporal Breakpoints for Multiverse Debugging.” In: *Proceedings of the 16th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2023, Cascais, Portugal, October 23-24, 2023*. Ed. by João Saraiva, Thomas Dagueule, and Elizabeth Scott. ACM, 2023, pp. 125–137. DOI: 10.1145/3623476.3623526.

- [158] Daniël A. A. Pelsmaeker, Hendrik van Antwerpen, Casper Bach Poulsen, and Eelco Visser. “Language-parametric static semantic code completion.” In: *Proc. ACM Program. Lang.* 6.OOPSLA1 (2022), pp. 1–30. DOI: 10.1145/3527329.
- [159] Daniël A. A. Pelsmaeker, Hendrik van Antwerpen, and Eelco Visser. “Towards Language-Parametric Semantic Editor Services Based on Declarative Type System Specifications (Brave New Idea Paper).” In: *33rd European Conference on Object-Oriented Programming, ECOOP 2019, July 15-19, 2019, London, United Kingdom*. Ed. by Alastair F. Donaldson. Vol. 134. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 26:1–26:18. DOI: 10.4230/LIPICSECOOP.2019.26.
- [160] Simon Peyton Jones, ed. *Haskell 98, Language and Libraries. The Revised Report*. Cambridge University Press, 2003.
- [161] Matthew Pickering, Nicolas Wu, and Csongor Kiss. “Multi-stage programs in context.” In: *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell, Haskell@ICFP 2019, Berlin, Germany, August 18-23, 2019*. Ed. by Richard A. Eisenberg. ACM, 2019, pp. 71–84. DOI: 10.1145/3331545.3342597.
- [162] J. F. Pimentel, L. Murta, V. Braganholo, and J. Freire. “A Large-Scale Study About Quality and Reproducibility of Jupyter Notebooks.” In: *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. 2019, pp. 507–517.
- [163] Gordon Plotkin and Matija Pretnar. “Handlers of Algebraic Effects.” In: *Programming Languages and Systems*. Ed. by Giuseppe Castagna. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 80–94. ISBN: 978-3-642-00590-9.
- [164] Gordon D. Plotkin. “A structural approach to operational semantics.” In: *The Journal of Logic and Algebraic Programming* 60-61 (2004), pp. 17–139. ISSN: 1567-8326.
- [165] Gordon D. Plotkin. “A structural approach to operational semantics.” In: *J. Log. Algebraic Methods Program.* 60-61 (2004), pp. 17–139.

- [166] Gordon D. Plotkin and John Power. "Adequacy for Algebraic Effects." In: *Foundations of Software Science and Computation Structures, 4th International Conference, FOSSACS 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings*. Ed. by Furio Honsell and Marino Miculan. Vol. 2030. Lecture Notes in Computer Science. Springer, 2001, pp. 1–24. DOI: 10.1007/3-540-45315-6_1.
- [167] Gordon D. Plotkin and Matija Pretnar. "Handlers of Algebraic Effects." In: *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*. Ed. by Giuseppe Castagna. Vol. 5502. Lecture Notes in Computer Science. Springer, 2009, pp. 80–94. DOI: 10.1007/978-3-642-00590-9_7.
- [168] Guillaume Pothier, Éric Tanter, and José Piquer. "Scalable omniscient debugging." In: *ACM SIGPLAN Notices* 42.10 (2007), pp. 535–552. DOI: 10.1145/1297105.1297067.
- [169] Guillaume Pothier, Éric Tanter, and José M. Piquer. "Scalable omniscient debugging." In: *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*. Ed. by Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr. ACM, 2007, pp. 535–552. DOI: 10.1145/1297027.1297067.
- [170] Casper Bach Poulsen, Pierre Néron, Andrew P. Tolmach, and Eelco Visser. "Scopes Describe Frames: A Uniform Model for Memory Layout in Dynamic Semantics." In: *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*. Ed. by Shriram Krishnamurthi and Benjamin S. Lerner. Vol. 56. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016, 20:1–20:26. DOI: 10.4230/LIPICS.ECOOP.2016.20.
- [171] Veselin Raychev, Martin T. Vechev, and Eran Yahav. "Code completion with statistical language models." In: *ACM*

- SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*. Ed. by Michael F. P. O'Boyle and Keshav Pingali. ACM, 2014, pp. 419–428. DOI: 10.1145/2594291.2594321.
- [172] Patrick Rein, Stefan Ramson, Jens Lincke, Robert Hirschfeld, and Tobias Pape. “Exploratory and Live, Programming and Coding.” In: *The Art, Science, and Engineering of Programming* 3.1 (2018). ISSN: 2473-7321. DOI: 10.22152/programming-journal.org/2019/3/1.
- [173] Patrick Rein, Stefan Ramson, Jens Lincke, Robert Hirschfeld, and Tobias Pape. “Exploratory and Live, Programming and Coding: A Literature Study Comparing Perspectives on Liveness.” In: *The Art Science and Engineering of Programming* (July 2018). DOI: 10.22152/programming-journal.org/2019/3/1.
- [174] Patrick Rein, Stefan Ramson, Jens Lincke, Robert Hirschfeld, and Tobias Pape. “Exploratory and Live, Programming and Coding - A Literature Study Comparing Perspectives on Liveness.” In: *Art Sci. Eng. Program.* 3.1 (2019), p. 1. DOI: 10.22152/programming-journal.org/2019/3/1.
- [175] Cas van der Rest and Casper Bach Poulsen. “Towards a Language for Defining Reusable Programming Language Components - (Project Paper).” In: *Trends in Functional Programming - 23rd International Symposium, TFP 2022, Virtual Event, March 17-18, 2022, Revised Selected Papers*. Ed. by Wouter Swierstra and Nicolas Wu. Vol. 13401. Lecture Notes in Computer Science. Springer, 2022, pp. 18–38. DOI: 10.1007/978-3-031-21314-4_2.
- [176] John C. Reynolds. “Definitional Interpreters for Higher-Order Programming Languages.” In: *Proceedings of the ACM Annual Conference - Volume 2*. Boston, Massachusetts, USA, 1972, pp. 717–740. ISBN: 9781450374927.
- [177] John C. Reynolds. “Definitional Interpreters for Higher-Order Programming Languages.” In: *Higher-Order and Symbolic Computation* 11.4 (1998), pp. 363–397.

- [178] John C. Reynolds. “Definitional Interpreters Revisited.” In: *Higher-Order and Symbolic Computation* 11.4 (1998), pp. 355–361.
- [179] Richard K. Ridgway. “Compiling routines.” In: *Proceedings of the 1952 ACM National Meeting (Toronto)*. ACM ’52. Toronto, Ontario, Canada: Association for Computing Machinery, 1952, pp. 1–5. ISBN: 9781450379250. DOI: 10.1145/800259.808980.
- [180] Philipp Riemer, Yury Nikulin, Ashley Claymore, and Mikhail Barash. “Optimal Language Design is Hard: A Case Study in ECMAScript (JavaScript) Standardization.” In: *Proceedings of the 18th ACM SIGPLAN International Conference on Software Language Engineering*. SLE ’25. Koblenz, Germany: Association for Computing Machinery, 2025, pp. 84–97. ISBN: 9798400718847. DOI: 10.1145/3732771.3742715.
- [181] Grigore Rosu and Traian-Florin Serbanuta. “An overview of the K semantic framework.” In: *J. Log. Algebraic Methods Program.* 79.6 (2010), pp. 397–434. DOI: 10.1016/j.jlap.2010.03.012.
- [182] Grigore Rosu and Traian-Florin Serbanuta. “K Overview and SIMPLE Case Study.” In: *Electron. Notes Theor. Comput. Sci.* 304 (2014), pp. 3–56. DOI: 10.1016/j.entcs.2014.05.002.
- [183] Arjen Rouvoet, Casper Bach Poulsen, Robbert Krebbers, and Eelco Visser. “Intrinsically-Typed Definitional Interpreters for Linear, Session-Typed Languages.” In: *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP’20)*. 2020, pp. 284–298. ISBN: 9781450370974. DOI: 10.1145/3372885.3373818.
- [184] Riemer van Rozen. “Languages of Games and Play: A Systematic Mapping Study.” In: *ACM Comput. Surv.* 53.6 (2021), 123:1–123:37. DOI: 10.1145/3412843.
- [185] Adam Rule, Aurélien Tabard, and James D. Hollan. “Exploration and Explanation in Computational Notebooks.” In: *Proceedings of the 2018 CHI Conference on Human Fac-*

- tors in Computing Systems*. CHI '18. Montreal QC, Canada: ACM, 2018, pp. 1–12. ISBN: 9781450356206.
- [186] Bruno C. d. S. Oliveira and William R. Cook. “Extensibility for the Masses - Practical Extensibility with Object Algebras.” In: *ECOOP 2012 - Object-Oriented Programming - 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings*. Ed. by James Noble. Vol. 7313. Lecture Notes in Computer Science. Springer, 2012, pp. 2–27. DOI: 10.1007/978-3-642-31057-7_2.
- [187] Jean E. Sammet. “Programming Languages: History and Future.” In: *Commun. ACM* 15.7 (1972), pp. 601–610. DOI: 10.1145/361454.361485.
- [188] David A. Schmidt. “Natural-Semantics-Based Abstract Interpretation (Preliminary Version).” In: *Static Analysis, Second International Symposium, SAS'95, Glasgow, UK, September 25-27, 1995, Proceedings*. Ed. by Alan Mycroft. Vol. 983. Lecture Notes in Computer Science. Springer, 1995, pp. 1–18. DOI: 10.1007/3-540-60360-3_28.
- [189] David A. Schmidt. “Data Flow Analysis is Model Checking of Abstract Interpretations.” In: *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, January 19-21, 1998*. Ed. by David B. MacQueen and Luca Cardelli. ACM, 1998, pp. 38–48. DOI: 10.1145/268946.268950.
- [190] David A. Schmidt. “Trace-Based Abstract Interpretation of Operational Semantics.” In: *LISP Symb. Comput.* 10.3 (1998), pp. 237–271.
- [191] David A. Schmidt and Bernhard Steffen. “Program Analysis as Model Checking of Abstract Interpretations.” In: *Static Analysis, 5th International Symposium, SAS '98, Pisa, Italy, September 14-16, 1998, Proceedings*. Ed. by Giorgio Levi. Vol. 1503. Lecture Notes in Computer Science. Springer, 1998, pp. 351–380. DOI: 10.1007/3-540-49727-7_22.
- [192] Elizabeth Scott and Adrian Johnstone. “GLL Parsing.” In: *Electron. Notes Theor. Comput. Sci.* 253.7 (2010), pp. 177–189. DOI: 10.1016/j.entcs.2010.08.041.

- [193] Elizabeth Scott and Adrian Johnstone. "GLL Parsing." In: *Electronic Notes in Theoretical Computer Science* 253.7 (2010). Proceedings of the Ninth Workshop on Language Descriptions Tools and Applications (LDTA 2009), pp. 177–189. ISSN: 1571-0661. DOI: 10.1016/j.entcs.2010.08.041.
- [194] John R Searle. *The construction of social reality*. Simon and Schuster, 1995.
- [195] Alejandro Serrano. "Type Error Customization for Embedded Domain-Specific Languages." PhD thesis. Utrecht University, Netherlands, 2018. URL: <http://dspace.library.uu.nl/handle/1874/363523>.
- [196] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strnisa. "Ott: Effective tool support for the working semanticist." In: *Journal of Functional Programming*. 20.1 (Mar. 22, 2010), pp. 71–122. DOI: 10.1017/S0956796809990293.
- [197] Tim Sheard and Simon L. Peyton Jones. "Template meta-programming for Haskell." In: *ACM SIGPLAN Notices* 37.12 (2002), pp. 60–75. DOI: 10.1145/636517.636528.
- [198] S. Doaitse Swierstra, Pablo R. Azero Alcocer, and João Saraiva. "Designing and implementing combinator languages." In: *Advanced Functional Programming*. Springer Berlin Heidelberg, 1999, pp. 150–206. DOI: 10.1007/10704973_4.
- [199] Wouter Swierstra. "Data Types à La Carte." In: *Journal of Functional Programming*. 18.4 (July 2008), pp. 423–436. ISSN: 0956-7968. DOI: 10.1017/S0956796808006758.
- [200] Rick Teuthof. "Truffle-Powered Execution of Component-Based Semantics." MA thesis. University of Amsterdam, 2025.
- [201] Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. "Languages as libraries." In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*. Ed. by Mary W. Hall and David A. Padua. ACM, 2011, pp. 132–141. DOI: 10.1145/1993498.1993514.

- [202] Masaru Tomita. *Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems*. Kluwer Academic Publishers, 1985. ISBN: 0898382025.
- [203] Cassia Torczon, Emmanuel Suárez Acevedo, Shubh Agrawal, Joey Velez-Ginorio, and Stephanie Weirich. “Effects and Coeffects in Call-by-Push-Value.” In: *Proc. ACM Program. Lang.* 8.OOPSLA2 (Oct. 2024). DOI: 10.1145/3689750.
- [204] Laurence Tratt. “Domain specific language implementation via compile-time meta-programming.” In: *ACM Trans. Program. Lang. Syst.* 30.6 (2008), 31:1–31:40. DOI: 10.1145/1391956.1391958.
- [205] J. Trenouth. “A Survey of Exploratory Software Development.” In: *The Computer Journal* 34.2 (Jan. 1991), pp. 153–163. ISSN: 0010-4620. DOI: 10.1093/comjnl/34.2.153. eprint: <https://academic.oup.com/comjnl/article-pdf/34/2/153/1400604/340153.pdf>.
- [206] J. Trenouth. “A Survey of Exploratory Software Development.” In: *The Computer Journal* 34.2 (Jan. 1991), pp. 153–163. ISSN: 0010-4620. DOI: 10.1093/comjnl/34.2.153.
- [207] Nicola Valcasara. *Unreal engine game development blueprints*. Packt Publishing Ltd, 2015. ISBN: 9781784397777.
- [208] L. Thomas Van Binsbergen. *GLL parser with simple combinator interface*. URL: <https://hackage.haskell.org/package/gll>.
- [209] L. Thomas van Binsbergen, Elizabeth Scott, and Adrian Johnstone. “Purely functional GLL parsing.” In: *Journal of Computer Languages* 58 (2020), p. 100945. ISSN: 2590-1184. DOI: <https://doi.org/10.1016/j.co-la.2020.100945>.
- [210] Eric Van Wyk, Oege de Moor, Kevin Backhouse, and Paul Kwiatkowski. “Forwarding in Attribute Grammars for Modular Language Design.” English. In: *Compiler Construction*. Ed. by R.Nigel Horspool. Vol. 2304. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2002, pp. 128–142. ISBN: 978-3-540-43369-9. DOI: 10.1007/3-540-45937-5_11.

- [211] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. "Attention is All you Need." In: *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*. Ed. by Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett. 2017, pp. 5998–6008.
- [212] Vlad A. Vergu, Pierre Neron, and Eelco Visser. "DynSem: A DSL for Dynamic Semantics Specification." In: *26th International Conference on Rewriting Techniques and Applications, RTA 2015*. Vol. 36. Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015, pp. 365–378. ISBN: 978-3-939897-85-9.
- [213] M Voelter and K Solomatov. *Language modularization and composition with projectional language workbenches illustrated with MPS*. *Software Language Engineering*. 2010.
- [214] Philip Wadler. "Monads for functional programming." In: *Program Design Calculi, Proceedings of the NATO Advanced Study Institute on Program Design Calculi, Marktoberdorf, Germany, July 28 - August 9, 1992*. Ed. by Manfred Broy. Vol. 118. NATO ASI Series. Springer, 1992, pp. 233–264. DOI: 10.1007/978-3-662-02880-3_8.
- [215] Philip Wadler et al. "The expression problem." In: *Posted on the Java Genericity mailing list (1998)*.
- [216] Michał Walicki and Sigurd Meldal. "Algebraic Approaches to Nondeterminism – an Overview." In: *ACM Computing Surveys* 29.1 (Mar. 1997), pp. 30–81. ISSN: 0360-0300.
- [217] Robert Michael Walsh. "Adapting Compiler Front Ends for Generalised Parsing." English. PhD thesis. Royal Holloway, University of London, 2016.
- [218] Chaozheng Wang, Junhao Hu, Cuiyun Gao, Yu Jin, Tao Xie, Hailiang Huang, Zhenyu Lei, and Yuetang Deng. "How Practitioners Expect Code Completion?" In: *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Soft-*

- ware Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*. Ed. by Satish Chandra, Kelly Blincoe, and Paolo Tonella. ACM, 2023, pp. 1294–1306. DOI: 10.1145/3611643.3616280.
- [219] Richard L Wexelblat. *History of programming languages*. Academic Press, 2014.
- [220] Man-Fai Wong, Shangxin Guo, Ching Nam Hang, Siu-Wai Ho, and Chee-Wei Tan. “Natural Language Generation and Understanding of Big Code for AI-Assisted Programming: A Review.” In: *Entropy* 25.6 (2023), p. 888. DOI: 10.3390/E25060888.
- [221] Nicolas Wu, Tom Schrijvers, and Ralf Hinze. “Effect Handlers in Scope.” In: *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*. Haskell 2014. ACM, 2014, pp. 1–12.
- [222] Eric Van Wyk. “Context in Parsing: Techniques and Applications.” In: *Eelco Visser Commemorative Symposium, EVCS 2023, April 5, 2023, Delft, The Netherlands*. Ed. by Ralf Lämmel, Peter D. Mosses, and Friedrich Steimann. Vol. 109. OASICS. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, 30:1–30:10. DOI: 10.4230/OASICS.EVCS.2023.30.
- [223] Burak Yetistiren, Isik Ozsoy, and Eray Tuzun. “Assessing the quality of GitHub copilot’s code generation.” In: *Proceedings of the 18th International Conference on Predictive Models and Data Analytics in Software Engineering, PROMISE 2022, Singapore, Singapore, 17 November 2022*. Ed. by Shane McIntosh, Weiyi Shang, and Gema Rodríguez-Pérez. ACM, 2022, pp. 62–71. DOI: 10.1145/3558489.3559072.
- [224] Aron Zwaan and Hendrik van Antwerpen. “Scope Graphs: The Story so Far.” In: *Eelco Visser Commemorative Symposium, EVCS 2023, April 5, 2023, Delft, The Netherlands*. Ed. by Ralf Lämmel, Peter D. Mosses, and Friedrich Steimann. Vol. 109. OASICS. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, 32:1–32:13. DOI: 10.4230/OASICS.EVCS.2023.32.

SUMMARY

Programming languages can affect the quality of software and the activity of writing the software. For example, Python is well suited for data science applications due to the immense number of data science libraries available for use in Python programs. However, Python is not suited for all applications. For instance, usage of Python in the domain of embedded system is less likely since memory and compute performance are not the main concerns in the design of the Python language. For such domains, a language closer to the hardware, such as C, is more applicable. Although Python and C are both general-purpose languages and not designed with a specific domain in mind, they do make different trade-offs in their design that make them suitable for different applications. There are also languages that are specifically designed for certain domains and using the language outside that domain is impractical. An example of such a language is \TeX , which is designed for the domain of typesetting documents. Choosing the right programming language for a problem is thus a consideration between different aspects, such as performance constraints, availability of third-party libraries, existing code to build on, expressivity of the language in the problem domain, etc. All these aspects can affect the quality of the software. For example, a high degree of expressivity can result in a shorter program and make maintenance of the software easier because the program is closer to the domain and there are less lines to maintain.

In some cases it is more interesting to design and implement a new language instead of using an existing language. However, developing a new programming language requires a significant design and engineering effort. Many design choices need to be made, and the right choice is not always obvious. This can be observed in many existing programming languages which continue to go through several high cost revisions. To aid the language design process, exploratory programming — a development style focused on variant creation and evaluation — can be used to explore the design space in an interactive and programmable manner. On the engineering side, making a programming language executable is already a significant effort. The effort is further increased by the myriad of auxiliary tools aiding developers, such as debuggers and integrations with modern editors. To reduce the engineering efforts, language workbenches can be used, which provide systematic methods for building (domain-specific) languages.

In this thesis we introduce exploratory language development: an approach that aims to improve the language creation process by combining features from language workbenches with a methodology focused on enabling exploratory programming for programming language design. The main goal of exploratory language development is to simplify the process of going from language idea to language prototype, experiment with this prototype, modify the original prototype to create new versions, and evaluate the different prototypes with the objective of obtaining insights that feedback into the design process.

We make four concrete contributions.

1. We design a reusable framework for exploratory programming.
2. We design a meta-language which is focused on variability in the language design process.
3. We design an approach that automatically derives auto-completion systems from the semantics of a programming language.
4. We define a reusable framework for debugging of non-deterministic programs.

The first contribution is foundational to the approach. The second contribution builds upon the first and focuses on the variability in the design process. The third and fourth contributions reduce engineering efforts by automatically deriving auxiliary tooling, which can improve the effectiveness of prototype evaluation.

Our first contribution was evaluated through case studies with a focus on the generality of the approach. Case studies were also used for the second contribution, to demonstrate the degree of variability and the expressiveness of the approach. For the third contribution, we built a benchmark to compare our approach to the state of the art in terms of precision and recall on suggested auto-completion candidates. And for the fourth contribution, we derived requirements from user stories collected through case studies. Using these requirements, we evaluated our approach on applicability and reusability compared to the state of the art.

SAMENVATTING

Programmeertalen kunnen effect hebben op de kwaliteit van software en de activiteit van het maken van de software. Neem bijvoorbeeld Python, een taal zeer geschikt voor een domein als data science door de grote hoeveelheid beschikbare libraries. Echter is Python niet geschikt voor alle domeinen. Een voorbeeld domein waar Python minder geschikt is is embedded systemen, aangezien berekenen en geheugen prestaties niet primair zijn in de design van Python. Voor zulke domeinen is een taal die dichter bij de hardware staat, zoals C, geschikter. Ondanks dat Python en C beide algemene programmeertalen zijn, maken ze verschillende afwegingen qua ontwerpkeuzes die ze meer of minder geschikt maken voor een bepaalde applicatie. Er zijn ook talen die specifiek ontworpen zijn voor een domein en daarbuiten minder geschikt zijn. Een voorbeeld van zo een taal is \TeX , welke is ontworpen voor het netwerk van documenten. Het kiezen van de juiste programmeertaal voor een probleem is dus een afweging, waarbij gekeken wordt naar verschillende aspecten van de taal, zoals het geheugengebruik van programma's in de taal, beschikbaarheid van bibliotheken, is de taal gebruikt voor bestaande software, hoe makkelijk is het om het domein uit te drukken in de taal, etc. All deze aspecten kunnen effect hebben op de kwaliteit van de geschreven software. Een taal die dicht bij het domein staat kan beheer van programma's eenvoudiger maken, maar ook het schrijven van een programma zelf, aangezien er minder code nodig is om een probleem te beschrijven.

In bepaalde gevallen is het interessanter om een nieuwe taal te ontwerpen en implementeren. Dit is echter geen eenvoudige stap en vereist een flinke investering in zowel ontwerp- als implementeerwerk. Er moeten namelijk veel ontwerpkeuzes worden gemaakt, en de juiste keuze is niet altijd eenvoudig. Dit is evident in bestaande programmeertalen, waar ontwerpkeuzes vaak herziend worden. Om het ontwerpproces te ondersteunen gebruiken wij explorerend programmeren. Dat is een stijl van programmeren met een focus het creëren en evalueren van verschillende varianten. Daarnaast is het implementatiewerk ook veeleisend. Het uitvoerbaar maken van een programmeertaal is namelijk veel werk. Het gebruiksvriendelijk maken van een programmeertaal, door extra tooling aan te beiden, is nog veel meer werk. Om dit te reduceren kan er gebruik worden gemaakt van language workbenches, welke een systematische manier aanbieden om talen sneller te ontwikkelen.

In dit proefschrift introduceren we *exploratory language development*, een methode die het maken van nieuwe programmeertalen probeert te vereenvoudigen. Dit doen wij door functionaliteit van language workbenches te combineren met een methode om explorerend programmeren mogelijk te maken voor taal implementatie. Het voornaamste doel van onze methodologie is om het eenvoudiger te maken om van taal idee naar taal prototype te gaan, dan experimenteren met dit prototype, andere variaties van de taal bouwen, en de verschillende variaties evalueren om inzicht te krijgen in ontwerpkeuzes. Hiervoor doen wij vier bijdragen aan de bestaande literatuur.

1. We presenteren een raamwerk voor exploreren programmeren, waarbij een taal automatisch dit type programmeren kan ondersteunen.
2. We ontwerpen en implementeren een nieuwe metataal met een focus op variabiliteit om het ontwerpproces te ondersteunen.
3. We ontwerpen een methodologie waarbij auto-completion services automatisch gegenereerd kunnen worden aan de hand van de formele semantiek van een taal.
4. We definiëren een herbruikbaar raamwerk voor het debuggen van niet deterministische programma's.

De eerste contributie is een basis waarop andere contributies bouwen, zoals de tweede contributie. De derde en vierde contributie verkleinen het implementatiewerk door automatisch tooling for programmeertalen te genereren aan de hand van de formele semantiek van de taal. Dit kan bijdragen aan het schrijven van programma's in de gedefinieerde taal, wat weer kan bijdragen aan het evalueren van het prototype.

Onze eerste contributie is geëvalueerd aan de hand van casussen met een focus op de algemeenheid van de aanpak. Voor de tweede contributie hebben we ook met casussen gewerkt om de mate van variabiliteit die ondersteund wordt door de aanpak te testen. Voor de derde contributie hebben een benchmark gemaakt van voorbeeld programma's om onze aanpak te vergelijken met bestaande methodes op gebied van recall en precision van de gegeven kandidaten. For de vierde contributie hebben we vereisten verzameld met behulp van een casussen. Met deze vereisten hebben we onze aanpak geëvalueerd en vergeleken met een bestaande aanpak op het gebied van toepasbaarheid en herbruikbaarheid.