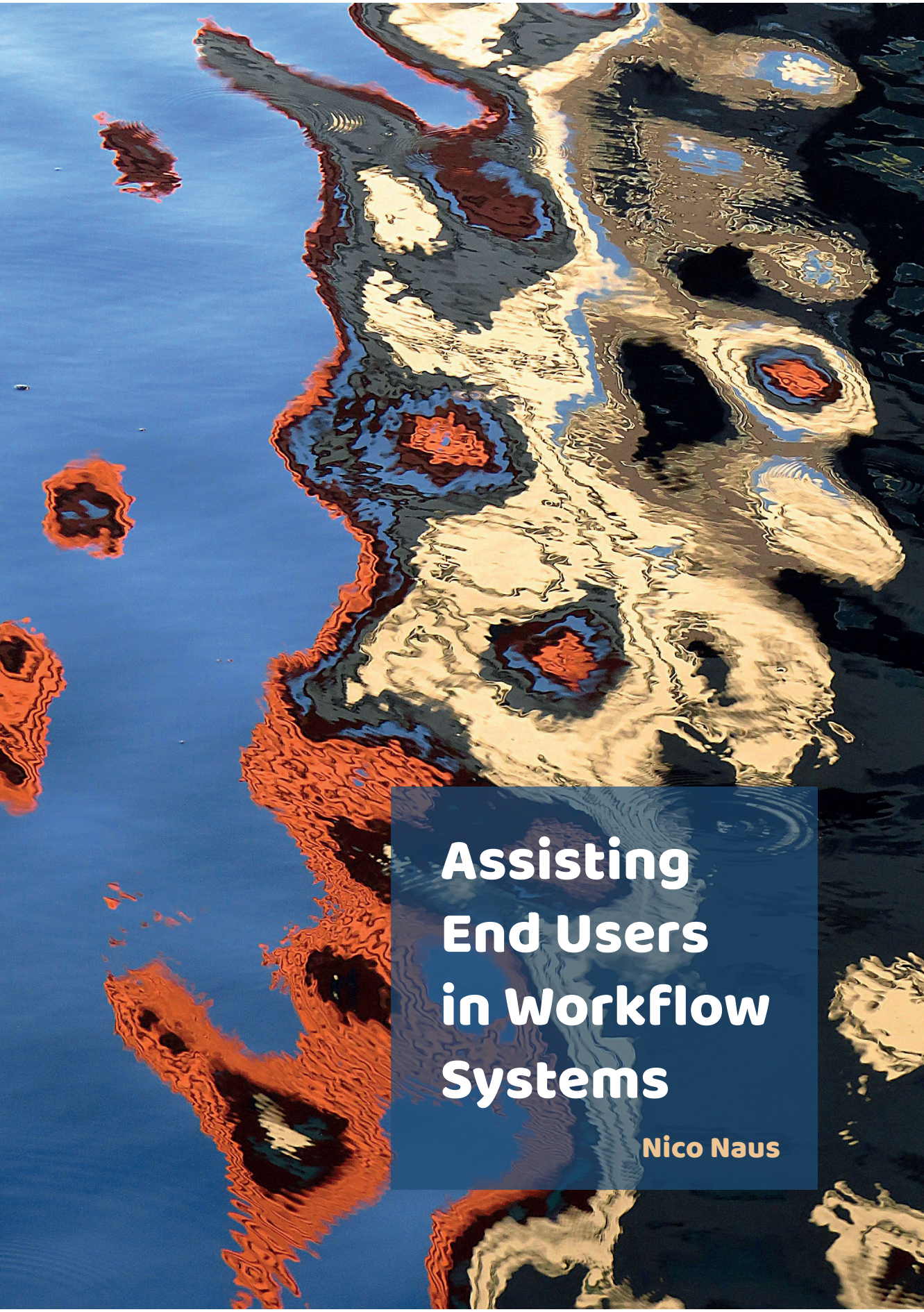




Assisting End Users in Workflow Systems

Nico Naus

Assisting End Users in Workflow Systems

Cover image: Jacqueline Hulst - luminosa.nl

Cover design: Joyce Vanhommerig - vanhommerig.be

ISBN: 978-90-393-7297-5

Assisting End Users in Workflow Systems

Ondersteuning voor gebruikers van Workflowsystemen
(met een samenvatting in het Nederlands)

PROEFSCHRIFT

*ter verkrijging van de graad van doctor aan de Universiteit Utrecht op
gezag van de rector magnificus, prof.dr. H.R.B.M. Kummeling, ingevolge
het besluit van het college voor promoties in het openbaar te verdedigen op
maandag 29 juni 2020 des middags te 12.45 uur*

door

*Nico Naus
geboren op 16 augustus 1991
te Gorinchem*

Promotor: Prof. dr. J.Th. Jeuring

"You have to take seriously the notion that understanding the universe is your responsibility, because the only understanding of the universe that will be useful to you is your own understanding."

Terence McKenna

Samenvatting

Tegenwoordig gebruikt bijna ieder bedrijf en iedere instelling workflow software voor hun werkprocessen. Ziekenhuizen gebruiken software om hun zorgprocessen te automatiseren. De kustwacht gebruikt workflowsystemen ter ondersteuning van zoek- en reddingsoperaties. Marineschepen gebruiken werkprocesautomatiseringssoftware om mensen, middelen en missiedoelen te beheersen.

Voor deze systemen hun intrede deden, kenden gebruikers de processen door en door, en waren ze zich er van bewust hoe hun keuzes het proces beïnvloeden. Workflowsoftware verbergt het verloop van de processen achter interfaces. Voor eindgebruikers is het nu niet altijd duidelijk hoe keuzes invloed uitoefenen op een taak. De informatie die beschikbaar is, speelt ook een rol in het beslissingsproces van een eindgebruiker. Hoe weet de gebruiker dat alle informatie in ogenschouw genomen is, voor een beslissing genomen wordt? Een manier om gebruikers van meer informatie te voorzien over hun huidige situatie, is door de gebruiker te voorzien van hints over de direct te nemen volgende stap. Deze hints zijn gebaseerd op de huidige situatie van de eindgebruiker: de positie in de werkstroom en de gegevens in het systeem. In deze dissertatie poog ik de vraag te beantwoorden, hoe we eindgebruikers kunnen voorzien van hints voor een volgende stap om ze te ondersteunen bij hun beslissingsproces.

Het antwoord op die vraag word in deze dissertatie gezocht in de toepassing van technieken ontwikkeld voor intelligente tutor systemen (ITSen) en automatische programma analyse. Eerder onderzoek naar ITS strategieën voor het oplossen van problemen in ITSen was de inspiratie voor de eerste aanpak om volgende-stap hints te genereren. Door het bestaande systeem uit te bereiden met extra informatie, kan het systeem gezien worden als een regel gebaseerd probleem, waardoor standaard AI zoekalgoritmes toegepast kunnen worden. De tweede aanpak is gebaseerd op technieken uit de automatische programma analyse. Door gebruik te maken van symbolische executie kunnen volgende-stap hints automatisch berekend worden, zonder dat de originele programmacode aangepast hoeft te worden.

Het toepassen van beide technieken resulteert in twee volgende-stap

hintsystemen. Het ene wordt ondersteund door de programmeur, het andere is volledig automatisch. Als onderdeel van de ontwikkeling van het automatische systeem wordt ook een formele taak-georiënteerd programmeren semantiek ontwikkeld, inclusief een symbolische executie semantiek. Beide systemen worden sound en complete bewezen. Ook zijn beide systemen geïmplementeerd, waarmee hun werking in de praktijk gedemonstreerd wordt. Het voorzien van gebruikers van volgende-stap hints is cruciaal in het verbeteren van de kwaliteit van beslissingen. Het helpt gebruikers om inzicht te krijgen in het effect van hun keuzes, en om er zeker van te zijn dat ze alle informatie in hun overweging mee hebben genomen.

Abstract

In today's society, almost every company and institution employs some kind of workflow automation. Hospitals employ software that automates health care processes. The coastal guard uses workflow software to assist in search and rescue operations. Naval ships use workflow automation software to manage people, resources and mission goals.

Before automation, users knew the process by heart, and knew how their choices influenced the process. Workflow systems hide the flow of processes behind interfaces. For end users, it is not always clear how decisions influence the progress of a task. Another factor in the decision process of an end user is the information available. How can a user be sure that he or she took all information into consideration before reaching a decision? One way to provide users with more information about their current situation is to provide them with next-step hints. These hints are based on their current situation: their position in the workflow and the data in the system. In this dissertation, I attempt to answer the question, how can we provide end users with next-step hints to aid them in making decisions?

The answer to that question is found by applying of techniques from intelligent tutoring systems (ITS) and program analysis. Previous work on ITS strategies inspired the first approach to generate next-step hints. By extending the original program with additional information, it can be viewed as a rule-based problem, making it susceptible to generic AI search and solving algorithms. The second approach comes from program analysis. By employing symbolic execution, next-step hints are automatically calculated, without any changes to the original code.

The application of both techniques results in two next-step hints systems. One system, aided by the programmer, the other fully automatic. In developing the automatic system, a formal Task-oriented Programming semantics is also developed, including a symbolic execution semantics. Both systems are proven to be sound and complete. They are both implemented, too, showing that they work in practice. Providing next-step hints to end users is crucial in improving the quality of decisions. It helps end users by giving insight into the effects of their choices, and makes sure that all data is taken into consideration.

Acknowledgements

I have always found it difficult to make choices in life. When the time came in high school to pick and choose courses, I chose all of them. After I finished high school, I was very unsure what to do next. Mathematics was always my strongest course in high school, so that was an obvious choice, or so I thought.

The first year of my mathematics bachelor was a tough one. My grades were not that good, and I felt a bit lost. After trying courses from several other programs, I became interested in Computer Science. I then completed my bachelors degree in computer science in three years. As the end of the program was nearing, I became more worried about the choice I had to make, what to do next? Most of my friends went off to industry, and I was wondering if I had to do the same. When my bachelor project was almost over, I knew I was not ready to end my time at the University just yet, so I applied for a Master.

I really enjoyed the courses on functional programming, compiler construction and program analysis, but what to choose as a topic for my master thesis? About the time I had to pick a topic, Peter Thiemann from Freiburg gave a presentation, suggesting several interesting Master thesis topics. These sparked my interest, and I went off to Freiburg to work on dynamic type inference for JavaScript. But, all good things come to an end, and again a choice had to be made, what to do next?

Over the years, I have noticed that I have several ways of dealing with choices. If possible, I try to do everything. When that is not an option, I'll try to postpone the decision as much as possible. And if that fails, making lists with pros and cons usually helps.

When the opportunity came up to do a PhD about making choices, the decision was quickly made! Over the course of my PhD, I investigated how we can use mathematics and computer science to help users of workflow systems make choices. I describe two different approaches to solving this problem in the coming chapters.

During my time at Utrecht University, I have also been able to pursue another interest of mine, namely politics. The opportunity came up to join a new University Council party called Utrecht PhD Party (UPP). I was lucky

to receive enough votes, so I could serve two years on the council. For one day a week, it was my job to collect as much information as possible, gather colleagues' opinions and weigh the consequences of new policies proposed by the board. I was able to improve the quality of the decisions made by the board and the council, and to improve life as a PhD candidate at the UU. At the same time, this was a remarkable opportunity to rapidly develop skills like understanding lengthy policy documents in relatively short time, negotiating a solution, and overcoming differences in opinions to find a solution that works for everyone. Looking back on my years as a council member now, it strikes me as funny that it was actually my job to make choices, while I feel that I struggle with exactly that.

After six years of studying at the university, four years of PhD research and two years as a council member, what have I learned in my personal quest for decision strategies? I think that can be summed up in one sentence.

With time, a solution will present itself.

I am very grateful for all the opportunities I have been given so far, especially the time to do research to obtain my PhD. And I am very much looking forward to the next challenge and adventure life will bring me.

First and foremost, I would like to thank my supervisor Johan Jeuring. He has given me the unique opportunity to develop myself as a researcher. Both his guidance and the freedom to pursue directions that I myself found interesting, stimulated me to better myself and my work.

Second, I would like to thank the Dutch Technology Foundation STW, which partly funded my research, together with NLDA, Chipsoft and the Netherlands Royal Navy. Thank you all for investing in the topic of Task-oriented Programming.

I have had more discussions than I can count with my colleagues Tim Steenvoorden and Markus Klinik from Radboud University. The many hours spend discussing ideas and solving problems any one of us had run into, were the main inspiration and driving force behind this dissertation.

I would like to thank Margo de Wolf, who has coached me on my presentation skills over the course of my PhD. Her many insightful suggestions allowed me improve my skills in conveying research results to others, in my opinion a crucial part of doing research.

This work is part of the research programme TOP Support for Collaborations on the Internet, with project number 13855, which is (partly) financed by the Netherlands Organisation for Scientific Research (NWO).

Work in the thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics)

Contents

Dutch summary	i
Abstract	iii
Acknowledgements	v
1 Introduction	1
1.1 Publications	4
1.2 Online resources	6
 I Rule-based problems & programmer assisted hint generation	 9
2 A multi-user feedback framework	11
2.1 Introduction	11
2.2 Problem description	13
2.2.1 Constructs	13
2.2.2 Hints	14
2.2.3 Research question	14
2.3 Problem formalisation	15
2.3.1 Semantics	17
2.4 Trace semantics	17
2.4.1 RuleTree observations	19
2.4.2 Traces of RuleTrees	20
2.5 Solving algorithms	21
2.5.1 Breadth First Trace	21
2.5.2 Heuristic Trace	22
2.6 Implementation	23
2.6.1 Properties of the traces function	24
2.6.2 Command & Control system	25
2.7 Conclusions	27
2.8 Related work	27

2.8.1	Rule-based problem modelling	27
2.8.2	Workflow Analysis	28
2.8.3	Decision Support Systems	29
2.8.4	Electronic Performance Support Systems	29
3	Generating next-step hints for tasks, puzzles and exercises	31
3.1	Introduction	31
3.2	Ideas	32
3.2.1	Using RuleTree to describe Ideas strategies	32
	Disjunctive Normal Form	33
	Gaussian Elimination	33
3.3	PuzzleScript	35
3.3.1	Solving Sokoban	36
3.4	iTasks	38
3.4.1	Solving a sliding puzzle	38
3.5	Conclusion	41
3.5.1	Ideas	41
3.5.2	PuzzleScript	42
3.5.3	iTasks	42
II	Task-oriented programming & automatic hint generation	43
4	An example-based introduction to task-oriented programming	45
4.1	Introduction	45
4.1.1	Task-oriented programming	45
4.1.2	Implementations of TOP	46
4.1.3	Challenges	47
4.2	TOP by example	47
4.2.1	Tasks model collaboration	47
4.2.2	Tasks are reusable	48
4.2.3	Tasks are driven by user input	48
4.2.4	Tasks can be observed	49
4.2.5	Tasks are never done	49
4.2.6	Tasks can share information	50
4.2.7	Tasks are predictable	51
4.3	Conclusion	51
5	TopHat	53
5.1	Introduction	53
5.2	Language	54
5.2.1	Expressions	54

5.2.2	Editors	56
5.2.3	Steps	58
5.2.4	Parallel	60
5.2.5	Annotations	61
5.3	Example	61
5.4	Semantics	63
5.4.1	Evaluating expressions	64
5.4.2	Task observations	66
5.4.3	Normalising tasks	68
5.4.4	Handling user inputs	70
5.4.5	Implementation	73
5.5	Properties	73
5.5.1	Type preservation	74
5.5.2	Progress	74
5.5.3	Soundness and completeness of Inputs	75
5.6	Related work	75
5.6.1	TOP implementations	75
5.6.2	Workflow modelling	76
5.6.3	Process algebras	77
5.6.4	Reactive programming	78
5.6.5	Session types	80
6	Symbolic TopHat	81
6.1	Introduction	81
6.2	Examples	82
6.2.1	Positive value	82
6.2.2	Tax subsidy request	83
6.2.3	Flight booking	85
6.3	Language	86
6.3.1	Expressions, values, and types	87
6.3.2	Inputs	88
6.3.3	Path conditions	89
6.4	Semantics	89
6.4.1	Symbolic evaluation	89
6.4.2	Observations	90
6.4.3	Normalisation	92
6.4.4	Handling	94
6.4.5	Simulating	94
6.4.6	Solving	98
6.4.7	Implementation	99
6.4.8	Outlook	99

6.5	Properties	100
6.5.1	Soundness	100
6.5.2	Completeness	101
6.6	Conclusion	102
6.6.1	Future work	103
6.7	Related work	103
6.7.1	Symbolic execution	103
6.7.2	Contracts	104
6.7.3	Axiomatic program verification	104
7	Assistive TopHat	107
7.1	Introduction	107
7.2	Examples	108
7.2.1	Tax subsidy request	108
7.2.2	Dining Computer Scientists problem	109
7.3	Generating next-step hints	110
7.3.1	Symbolic execution	111
7.3.2	Symbolic semantics	112
7.3.3	Next-step hints observation	113
7.3.4	Tax subsidy request	114
7.3.5	Dining Computer Scientists	115
7.4	Properties	116
7.4.1	Correctness of simulate	117
7.4.2	Correctness of hints	120
7.5	Related work	121
7.5.1	Automatic hint generation in intelligent tutoring systems	122
III	Conclusions	123
8	Conclusion	125
	What is the essence of the task-oriented programming paradigm?	125
	How can we define and guarantee properties of tasks?	126
	How can we calculate next-step hints from a work-flow specification?	126
9	Future work	129
9.1	End-user run-time feedback	129
9.1.1	Unified hints framework	129
9.1.2	iTasks integration	130

9.1.3	Hint presentation	130
9.1.4	Testing the effect of hints	130
9.1.5	Other kinds of feedback	131
9.2	Task analysis	131
9.2.1	Analysis of TopHat programs	131
9.2.2	Verification of iTasks behaviour	131
9.2.3	Workflow mining	132
9.3	TOP language development	133
9.3.1	Visual TopHat	133
9.3.2	TopHat 2.0	133
Curriculum Vitae		135
Appendices		139
A TopHat type preservation proofs		139
A.1	Type preservation under evaluation	140
A.2	Type preservation under striding	140
A.2.1	Task value preserves types	140
A.2.2	Striding preserves types	142
A.3	Proof of type preservation under normalisation	144
A.4	Proof of type preservation under handling	144
B TopHat progress proof		147
C Proof of correctness of Inputs function		151
D Symbolic TopHat soundness and completeness		155
D.1	Soundness proofs	156
D.1.1	Proof of soundness of symbolic evaluation semantics	156
D.1.2	Proof of soundness of symbolic striding semantics	160
D.1.3	Proof of soundness of symbolic normalisation semantics	162
D.1.4	Proof of soundness of symbolic handling semantics	163
D.1.5	Proof of soundness of symbolic interacting semantics	166
D.2	Completeness proofs	167
D.2.1	Proof of completeness of the symbolic handling semantics	167
D.2.2	Proof of completeness of the symbolic interaction semantics	169

E	Assistive TopHat soundness and completeness	171
E.1	Completeness proofs	172
E.1.1	Completeness of Simulate	172
E.1.2	Completeness of interaction	172
E.1.3	Completeness of handling	173
E.1.4	Completeness of normalisation	175
E.1.5	Completeness of striding	176
E.1.6	Completeness of evaluate	178
E.2	Soundness proofs	182
E.2.1	Soundness of simulate	182
E.2.2	Soundness of interaction	182
E.2.3	Soundness of handle	183
E.2.4	Soundness of normalise	185
E.2.5	Soundness of stride	186
E.2.6	Soundness of evaluate	188
E.2.7	$\mathcal{V}$ preserves consistency	191
	References	195

For Ype

Chapter 1

Introduction

Today's work environment looks radically different from what it was 20 years ago. In every field, technology is being used to guide business processes. It does not matter if the business is small or large, government or commercial, agriculture or services. Business process automation is omnipresent.

The workflow software industry is booming. Many frameworks exist, and it is a multi billion dollar industry.

Automation of business processes in workflow software has enormous benefits. It allows companies to analyse their processes and optimise them. Automation improves quality, and can contain cost.

But, automation also comes at a cost. Users need to be properly trained in order to use the automatic systems. Processes that were once known by heart, are now hidden behind user interfaces and buried in workflow systems. Choices that users make can impact the goal that they want to achieve in ways that they cannot oversee. They might feel that they are unable to oversee all information in the system.

In an attempt to overcome these problems, a business may employ a performance support system (PSS). The goal of a PSS is to facilitate so called just in time training on the job. When a worker requires information or training to perform a task, some tools are provided to obtain the required knowledge or skill, just in time (Schaik, Pearson, & Barker, 2002). A PSS enables workers to learn as they do, and improves their individual performance. Drawbacks of a PSS are that they do not help with the decisions a user has to take, only guidelines and general information is provided. A PSS does not take the current state of the system into account.

An alternative solution could be to use a decision support system (DSS). Such a system employs some kind of model of the problem that needs a decision. Then based on specific information, some kind of analysis is performed, to suggest a choice to the user (Shim et al., 2002).

Supporting users in their decision-making process has several benefits (Power, 2002). The individual productivity is improved. Users spend less time on the manipulation of data and on the clerical aspects of their job. The quality and speed of decisions increases. A DDS encourages fact-based decision making and saves time in retrieving decision relevant information. Workers improve their decision-making skills. Users develop a better understanding of the business and the environment in which they make decisions. On top of that, they require less expertise from users to perform their tasks.

Using a DSS has several potential drawbacks. Since it relies on a model of the problem, a DSS is very rigid. It requires developers to make a model of the problem, and if the workflow system that it supports is extended, the model needs to be updated as well. Furthermore, developing a DSS requires a significant financial investment (Power, 2002).

In this thesis I develop technologies that can be used to calculate next-step hints for end users, without the drawbacks of traditional DSSs. Based on the flow of the program, and the information in the system, we want to calculate next-step hints. By employing techniques from programming technology, formal methods and intelligent tutoring systems in a novel way, end users can be assisted in their work. The solutions presented require minimal to no effort from the programmer, mitigating the major drawbacks listed above. Validating the effectiveness of these technologies by means of user studies is left as future work.

How can we calculate next-step hints from a workflow specification?

We investigate this question using the Task-oriented Programming paradigm (TOP) as a basis.

TOP is a programming paradigm that aims to provide a high level of abstraction to programmers of workflow programs, while still being expressive enough to model real-world collaboration. The paradigm has been around for more than a decade, but only exists in implementation. The most used TOP implementation is iTasks (Plasmeijer, Lijnse, Michels, Achten, & Koopman, 2012). It has been subject of study for many years (P. Koopman, Lubbers, & Plasmeijer, 2018; P. W. M. Koopman, Plasmeijer, & Achten, 2008; Oortgiese, van Groningen, Achten, & Plasmeijer, 2017; Plasmeijer et al., 2011; Stutterheim, Achten, & Plasmeijer, 2017), and has been used to implement several systems for the Netherlands Royal Navy and the Dutch coast guard (Lijnse, Jansen, & Plasmeijer, 2012; Stutterheim, Achten, & Plasmeijer, 2016).

Part I describes how ideas from domain-specific languages and artificial intelligence can be applied to workflow systems to generate next-step hints.

In Chapter 2, a domain-specific language that allows the uniform description of rule-based problems is presented, together with a solving framework that generates next-step hints. This framework uses search algorithms from artificial intelligence to calculate traces to goal states. From these traces, next-step hints are generated. Both a formal system and an implementation in Haskell are developed. The formal solving framework is proven to be sound and complete. The framework is applied to an example program that is modelled after an iTasks implementation developed for the Netherlands Royal Navy (Stutterheim et al., 2016). Then in Chapter 3, we take a look at several example applications of this framework.

Part II describes a completely different approach to next-step hints, based on techniques from programming languages, semantics and formal methods. The fact that TOP only exists in implementation makes (formal) reasoning about TOP software harder.

To better equip ourselves to answer the question of how we can calculate next-step hints for TOP programs, we first take a step back and look at the paradigm itself.

What is the essence of the task-oriented programming paradigm?

Chapters 4 and 5 answer this question by exploring the TOP concepts first by example, and then by presenting a formal semantics for it, called TopHat ($\widehat{\text{TOP}}$). The language is embedded in the simply typed λ -calculus, and consists of a layered operational semantics. Evaluation of $\widehat{\text{TOP}}$ programs is driven by user input. By embedding the task language, it is clearly separated from the semantics of the underlying host language. Along with the semantics, the following observations over tasks are presented: the current value, whether a term is stuck, and the accepted inputs. Progress and type preservation properties are proven for $\widehat{\text{TOP}}$, and the whole semantics has been implemented in Haskell. An extensive comparison of $\widehat{\text{TOP}}$ with work in related areas is provided.

This formal semantics forms the ideal starting point for formal reasoning. Before coming back to the question of how we can calculate next-step hints, we first take a look at TOP programs themselves.

How can we define and guarantee properties of tasks?

Chapter 6 develops a symbolic execution semantics for $\widehat{\text{TOP}}$, called Symbolic $\widehat{\text{TOP}}$. The symbolic execution semantics is developed by modifying the original $\widehat{\text{TOP}}$ semantics in such a way that it can simulate the execution of a workflow without having actual user input. This allows us to gather all possible symbolic outcomes of a program, and then prove that it adheres to

a predefined specification. The symbolic execution semantics is shown to be sound and complete, and an implementation in Haskell is presented.

Coming back to the question of calculating next-step hints for workflow systems, we now find ourselves with all the ingredients to construct an automatic feedback framework. Chapter 7 presents this framework, called Assistive $\widehat{\text{TOP}}$. By taking a user defined goal, we can filter out all symbolic executions that we are interested in. Then these runs can be used to calculate next-step hints. We prove task simulation to be correct, and show that the next-step hints generated by Assistive $\widehat{\text{TOP}}$ are sound and complete.

Part III concludes the dissertation. The findings from Parts I and II are summed up in Chapter 8. Ideas for future work are presented in Chapter 9. The proofs mentioned in Parts I and II are then given in Appendices A to E.

1.1 Publications

The following publications were written over the course of my PhD.

2016

Building a Generic Feedback System for Rule-Based Problems

Nico Naus and Johan Jeuring. Trends in Functional Programming.

Together with Johan Jeuring, we explore how to build a generic feedback system that can be applied to several rule-based systems. We are interested in providing end users with feedback that guides them to their goal, without programmers needing to change much in their implementation. To do this, we propose to use a tree structure that lies on top of the original problem. This problem could be a game, a workflow or an intelligent tutoring system, to name a few examples.

All research is performed on my own, with regular discussions with my supervisor Johan Jeuring. The results of this paper are used in Chapters 2 and 3.

2019

TopHat: A formal foundation for task-oriented programming

Tim Steenvoorden, Nico Naus and Markus Klinik. Principles and Practice of Declarative Programming.

My colleagues Tim Steenvoorden, Markus Klinik and I have all been working on the subject of Task-oriented Programming (TOP). The techniques and ideas we wanted to apply to this subject often called for a formal

semantics. This had never been done, so we put it upon ourselves to develop a formal TOP semantics.

My contribution to this research started by discussing many different alternative designs with Tim Steenvoorden and Markus Klinik. I have been involved in the development of the formal semantics. Furthermore, I have written the related work section on workflow modelling. My biggest contribution was showing type preservation, progress, soundness and completeness for this system. The results of this work are used in Chapters 4 and 5.

A symbolic execution semantics for TopHat

Nico Naus, Tim Steenvoorden and Markus Klinik. Implementation and Application of Functional Languages.

To demonstrate that having a formal TOP semantics indeed allows for the application of interesting techniques, we set out to prove properties over $\widehat{\text{TOP}}$ programs. Our initial idea was to use Hoare logic, but to no avail. I came up with the alternative approach of using symbolic execution. This required alterations to the semantics. We had many discussions about what this new semantics should look like. When we had decided what a suitable solution would be, Tim Steenvoorden and I developed the formal semantics. The simulation function was developed by me, and Tim Steenvoorden was responsible for the implementation. Tim Steenvoorden wrote the examples and Markus Klinik the related work. I have proven the symbolic system to be sound and complete. The results of this work are used in Chapter 6.

2020

Generating next-step hints for task-oriented programs using symbolic execution

Nico Naus and Tim Steenvoorden. Trends In Functional Programming.

Coming back to the overall topic of providing end users with next-step hints, the idea was born to apply the symbolic execution which we had already developed. We allow users to set a goal condition, and filter out symbolic executions that interest them.

Developing the implementation was a joint effort. The correctness proof, and most of the writing has been done by me. Tim Steenvoorden greatly improved the narrative and made several improvements to the paper text. The introduction and abstract was a joint effort. The results of this work are used in Chapter 7.

Forthcoming

TopHat 2.0: An even more stylish programming language

Tim Steenvoorden and Nico Naus. To be submitted.

Working with $\widehat{\text{TOP}}$ on the above mentioned topics and several other topics of interest to Markus Klinik and Tim Steenvoorden provided us with many ideas on how the language could be improved. Research on the improvements is led by Tim Steenvoorden, also in discussion with Markus Klinik and myself. This publication is forthcoming, and its results are not mentioned in this dissertation. In Section 9.3.2, the improvements we have in mind are listed.

Multi User Generic Feedback in iTasks

Nico Naus. To be submitted.

Based on previous research on rule-based problem solving, a new version of the framework has been developed that supports multiple users, and is geared more towards iTasks. Results from this work are used in Chapter 2.

1.2 Online resources

The following online resources belonging to this dissertations and the publications that are part of it, are available.

rule-tree-semantics The Haskell implementation described in Chapter 2 can be found in the rule-tree-semantics GitHub repository.
<https://github.com/niconaus/rule-tree-semantics>

iTasks-feedback An alternative implementation in Clean is also available. This implementation is described in Chapter 3, and can be found in the iTasks-feedback GitHub repository.
<https://github.com/niconaus/iTasks-feedback>

tophat-haskell The $\widehat{\text{TOP}}$ implementation in Haskell implementation described in Chapter 5 can be found in the tophat-haskell GitHub repository.
<https://github.com/timjs/tophat-haskell>

tophat-clean The $\widehat{\text{TOP}}$ implementation on top of iTasks that is described in Chapter 5 can be found in the tophat-clean GitHub repository.

<https://github.com/timjs/tophat-clean>

symbolic-tophat-haskell The Symbolic $\widehat{\text{TOP}}$ and Assistive $\widehat{\text{TOP}}$ implementation described in Chapters 6 and 7 can be found in the symbolic-tophat-haskell GitHub repository.

<https://github.com/timjs/symbolic-tophat-haskell>

Part I

Rule-based problems & programmer assisted hint generation

Chapter 2

A multi-user feedback framework

Workflow systems are more and more common due to the automation of business processes. The automation of business processes enables organisations to simplify their processes, improve services and to contain costs. A problem with using workflow systems is that processes once known by heart, are now hidden from the user. This, combined with time pressure, lack of experience and an abundance of options, makes it harder for a user to make the right choices. To aid users of these systems, we have developed a multi-user rule-based problem-solving framework that can be instantiated for many workflow systems. It provides hints to the end user on how to achieve her goals and makes life for the programmer easier, as she only needs to instantiate the framework instead of programming an ad-hoc solution. Our approach consists of two parts. First, we present a domain-specific language (DSL) that offers commonly used constructs for combining components of different rule-based problems. Second, we use generic search algorithms to solve various kinds of problems. We show a practical implementation with an example workflow system. We show that this system fulfils several desirable properties.

2.1 Introduction

Due to the automation of business processes, more and more workflow systems are being used to manage and perform tasks. The Dutch coastal guard uses a workflow system to monitor the seas and to aid in emergencies (Lijnse et al., 2012). Hospitals use systems like EPIC or WebPT to manage patients, assign tasks and monitor treatment. Teachers use intelligent tutoring systems to allow students to receive immediate and personalised feedback on their exercises.

A downside of using workflow systems is that a process that was once known by heart, is now hidden from the user. This, combined with the fact

that there might be time pressure, lack of experience or an abundance of options, makes it harder for end users to make the right choices between the different options, to achieve the goal a user has in mind.

To overcome this problem, we want to assist a user in reaching her goals more efficiently.

This is commonly done by employing a decision support system (DSS) (Shim et al., 2002). Many different types of DSS exist, but they have several components in common. A DSS has some model that represents the domain in which a decision needs to be made. Using data about the current situation, together with the model, some kind of analysis is performed. The specific analysis used differs per DSS. Based on the results of the analysis, the DSS suggests a decision to the user.

Using a DSS has many advantages (Power, 2002). The productivity of individual users is improved. Users spend less time on the administrative aspects of the tasks they need to perform, and spend less time manipulating data. The quality and speed of the decisions is increased. Time spent on retrieving decision relevant information is reduced, and fact-based decision making is stimulated.

Traditional DSS have several downsides. First of all, since a DSS relies on a model of the problem, these systems are very rigid. If the problem is not modelled, the DSS cannot be used. When the problem or the domain are altered or expanded, a programmer needs to go back and change the model accordingly. A second downside is the large financial investment that is required (Power, 2002).

To overcome the downsides of a DSS, while still being able to enjoy its benefits, we present a multi-user rule-based problem solver. Our system consists of two parts; a domain-specific language (DSL) that allows programmers to express multi-user rule-based problems, and several generic solving algorithms that calculate traces to the goal, from which hints can be produced.

The advantage of our system is that it is much easier to implement a model that describes the multi-user rule-based problem. On top of that, once the model has been described, there is no need to develop a custom analysis. Once the model has been expressed in our DSL, one of the generic solving algorithms can be used to find a solution.

This chapter presents both a formal multi-user rule-based problem-solving framework, as well as a practical implementation. The trace semantics of the framework is shown to be sound and complete with respect to the regular semantics of our DSL, using a property verification tool.

Chapter 3 presents several other applications of the rule-based problem-solving framework, to show that it is indeed applicable to a wide variety of

problems.

2.2 Problem description

Our goal is to describe a generic framework for autonomously generating hints that end-users of workflow systems can use to achieve their goal(s). By a workflow system we mean a system that automates workflows, allows multiple users to collaborate, and works on some kind of shared data.

Van der Aalst et al. (Aalst, ter Hofstede, Kiepuszewski, & Barros, 2003) have identified common patterns of workflow systems. We will use this set to specify constructs used in workflow systems.

2.2.1 Constructs

The following constructs are common in most workflow systems.

Sequence	Perform activities one after the other.
Parallel Split	Perform multiple activities at the same time.
Exclusive Choice	Choose exactly one activity from a list.
Milestone	Make an activity available when the state is in a certain condition.
Interleaving	Perform activities in an arbitrary order.
Multi-Choice	Choose one or more activities from a list.
Arbitrary Cycles	Repeat part of a workflow an arbitrary number of times.

In traditional workflow systems, steps can pass data along to the next step, as well as work on shared data. We simplify our model of workflow systems to only consider shared data. Therefore we do not need constructs like synchronisation points and discriminators, as described by van der Aalst et al. Steps can immediately observe the result of every other step through the shared data, instead of having to wait on incoming branches.

Additionally, we want to support multiple users. The most straightforward way to accommodate this is by means of an Assign construct, which assigns an activity to a user or possibly a set of users. Such a construct is heavily used in for example the iTasks workflow framework (Plasmeijer et al., 2012).

2.2.2 Hints

The purpose of our framework is to give hints to end-users of workflow systems; information that they can use to achieve their goal. What is the best treatment to select for a certain patient? What action needs to be taken when a fire breaks out on a ship?

We use traces consisting of sequences of steps that lead users to states in which the goal has been reached. From these traces, richer feedback information can be constructed. For example, next-step hints can be constructed by returning the first element in the trace.

The traces that we want to generate are composed of sets of steps per time unit, where multiple users can perform a step in each time unit. Fig. 2.1 lists an example of such a trace, where $state_i$ is the state at time i .

Application of all steps in one time unit leads to the next state in the trace.

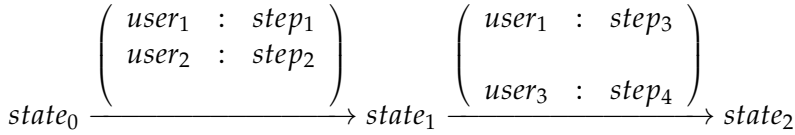


FIGURE 2.1: An example of what a trace might look like

For this to work properly, we require that all steps performed in one time unit are independent of each other. This means that the order of applying steps to the original state does not affect the resulting state. On top of that, we require that every user performs at most one step per time unit.

2.2.3 Research question

In the following sections, we will answer the question: how do we, for any given multi-user workflow problem, calculate traces that lead to a solution state?

We aim to answer this question by first tackling the issue of dealing with different multi-user workflow systems. By defining a domain-specific language that allows for the uniform description of problems, we can treat each of them in a similar manner. Then, to calculate the partial traces, we employ search algorithms from artificial intelligence.

2.3 Problem formalisation

A multi user workflow problem can be considered a well-defined artificial intelligence (AI) problem (Russell & Norvig, 2010), which consists of the following components.

Initial state	The state of the problem that you want to solve.
Operator set	The set of steps that can be taken, together with their effects.
Goal test	A predicate that is True if the problem is solved.
Path Cost function	A function that describes the cost of each operation.

This is similar to workflows: the state of the workflow system is the initial state, the steps users can take are the operator set, the goal the user has in mind is the goal test and finally the resources a workflow uses can be captured in a path cost function.

By choosing a uniform way to describe workflow problems as well-defined AI problems, we can treat each problem within the same framework.

Currently, several languages exist to allow programmers to describe workflow and rule-based problems, but none of them are completely suitable for our purposes. Workflow languages allow programmers to model complex behaviour that makes calculating a path to the goal of a user very complex or even unfeasible. They are therefore not suitable for our purposes. Existing rule-based problem modelling languages like PDDL (McDermott et al., 1998), STRIPS (Fikes & Nilsson, 1971), SITPLAN (Galagan, 1979) and PLANNER (Hewitt, 1969) have limitations that prevent us from fully describing the problems from the workflow domain. These languages do not support higher order definitions, and most of them only support a finite state-space. Therefore, we design our own rule-based problem modelling language.

We opt for a domain-specific language (DSL) that is embedded in a language that supports higher order programming. This means that our DSL is expressed in a standard programming language, called the host language. Embedding a DSL into a host language has the specific advantage that we can use all features from the host language in specifying programs in our DSL. In this case, we are particularly interested in using abstraction and application from the host language.

Figure 2.2 lists the components of our DSL.

RuleTree a	CR a	=	Cond (Pred a) (CR a)
= Seq [RuleTree a]			Rule n (Effect a)
Choice [RuleTree a]			$u @$ (Rule n (Effect a))
Par [RuleTree a]	Pred a	=	$a \rightarrow \text{Bool}$
Assign u (RuleTree a)	Effect a	=	$a \rightarrow a$
Leaf (CR a)	Goal a	=	$a \rightarrow \text{Bool}$
Empty	n	$\in$	set of names
	u	$\in$	set of user identifiers

FIGURE 2.2: Syntax of our rule-based problem DSL

With the DSL, we want to cover all workflow constructs mentioned in Section 2.2.1, as well as the elements of a well-defined AI problem as listed above.

We use a slightly simplified definition of an AI problem however. If there is a cost associated with a certain operation, we encode this as an effect on the state. This is the common way to encode these effects in workflow systems. Therefore, we do not need a path cost function.

The (initial) state is modelled by a value of type a .

The operator set is represented by a tree structure we call a RuleTree. This tree structure describes how operations, which we call rules, relate to each other. There are four ways to combine RuleTrees; in sequence (Seq), by choosing among them (Choice), in parallel (Par), or by assigning them to a user (Assign). These correspond to the workflow constructs sequence, exclusive choice, parallel split, and user assignment. The Milestone construct is modelled by means of a condition. Interleaving and multi-choice can be built from these constructs. For arbitrary cycles we rely on the host language to provide abstraction and application.

The design of the RuleTree is loosely based on strategy language from the Ideas framework (Heeren, Jeuring, & Gerdes, 2010), iTask combinators (Plasmeijer et al., 2012), and the strategy language presented by Visser and others (Visser, Benaissa, & Tolmach, 1998).

Finally, the goal test is represented by the predicate Goal a . These three components make up our DSL for describing rule-based problems.

The leaves of the RuleTree are CR a and Empty. Here, CR a is either a Cond or an actual rule, where the rule can be assigned to a user u ($u @ (\text{Rule } n (\text{Effect } a))$) or unassigned ($\text{Rule } n (\text{Effect } a)$), with n the name of the rule and Effect a the effect of the rule on the state. Conds can be nested. Rules can be seen as steps, tasks or the smallest units in which work can be divided.

Conditions are part of the leaves, and guard a CR a , which may contain another condition. A single leaf is considered to be an atomic action. This

prevents conflicts between rules and conditions when leaves are executed in parallel.

We implement the DSL as an embedded DSL in Haskell. This allows us to use standard Haskell functions to construct for example a `RuleTree`. We chose not to implement recursion in our DSL, but instead make use of recursion in the host language. The advantage of this is that we can keep our DSL simple and small. Implementing recursion in the DSL requires adding abstraction and application, making the DSL significantly more complex. Most rule-based problem can be encoded in this DSL, and as long as there is an appropriate solving algorithm available, our framework can generate hints for it.

2.3.1 Semantics

Figure 2.3 defines what it means to apply an entire `RuleTree` to a state. The result of the application is a set of end states that can be reached.

Application of a `RuleTree` is rather straightforward, except for the `Seq` and `Par` cases. If an error occurs inside a `Seq`, denoted by $\bot$, the whole sequence needs to be aborted since the next step does not become available. This can occur when a condition does not hold, or when a choice has to be made out of zero elements.

We are only interested in the final states that can be reached, not in the intermediate states. As a consequence, we can view the semantics of `Par` as interleaving of the individual steps contained in the sub-trees. The function `step` takes a `RuleTree` and calculates a set of tuples containing all steps that can be applied at this point and the remaining `RuleTree`. This result is then used by the `RuleTree` application to interleave all possible steps, and calculate the final state.

2.4 Trace semantics

We are not so much interested in the final state that is reached, but rather in the steps that users can take to transition between states. To calculate these steps, we use a trace semantics. The trace semantics consists of two parts, namely the firsts and empty observations over `RuleTrees`, and the function traces that makes use of these observations.

We introduce two new constructs that will be used to define the two parts.

$$\begin{aligned}
& \cdot : \text{RuleTree } a \times a \rightarrow \mathcal{P}(a) \\
& (\text{Seq } (rt:rts)) \cdot s \quad \begin{array}{l} | \quad rt \cdot s = \perp \mapsto \perp \\ | \quad rt \cdot s \neq \perp \mapsto \{x \mid s' \in rt \cdot s, x \in (\text{Seq } rts) \cdot s'\} \end{array} \\
& (\text{Seq } []) \cdot s = \{s\} \\
& (\text{Choice } (rt:rts)) \cdot s = rt \cdot s \cup (\text{Choice } rts) \cdot s \\
& (\text{Choice } []) \cdot s = \perp \\
& (\text{Par } (rt:rts)) \cdot s = \{(\text{Par } (rt':rts)) \cdot (r \cdot s) \mid (r, rt') \in \text{step } rt\} \\
& \quad \cup \{(\text{Par } (rt:rts')) \cdot (r \cdot s) \mid (r, rts') \in \text{step } (\text{Par } rts)\} \\
& (\text{Par } []) \cdot s = \{s\} \\
& (\text{Assign } u \text{ } rt) \cdot s = rt \cdot s \\
& (\text{Leaf } (\text{Cond } p \text{ } r)) \cdot s \quad \begin{array}{l} | \quad \neg ps \mapsto \perp \\ | \quad ps \mapsto (\text{Leaf } r) \cdot s \end{array} \\
& (\text{Leaf } (\text{Rule } n \text{ } e)) \cdot s = \{e \cdot s\} \\
& \text{Empty} \cdot s = \{s\}
\end{aligned}$$

(A) RuleTree application definition

$$\begin{aligned}
& \text{step} : \text{RuleTree } a \rightarrow \mathcal{P}(\text{RuleTree } a \times \text{RuleTree } a) \\
& \text{step } (\text{Seq } (rt:rts)) = (\cup \{\text{step } (\text{Seq } rts) \mid (\text{Empty}, \text{Empty}) \in \text{step } rt\}) \\
& \quad \cup \{(r, \text{Seq } (rt':rts)) \mid (r, rt') \in \text{step } rt\} \\
& \text{step } (\text{Seq } []) = \{(\text{Empty}, \text{Empty})\} \\
& \text{step } (\text{Choice } (rt:rts)) = \text{step } rt \cup \text{step } (\text{Choice } rts) \\
& \text{step } (\text{Choice } []) = \emptyset \\
& \text{step } (\text{Par } (rt:rts)) = (\cup \{\text{step } (\text{Par } rts) \mid (\text{Empty}, \text{Empty}) \in \text{step } rt\}) \\
& \quad \cup \{(r, \text{Par } (rt':rts)) \mid (r, rt') \in \text{step } rt\} \\
& \quad \cup \{(r', \text{Par } (rt:rts')) \mid (r', \text{Par } rts') \in \text{step } (\text{Par } rts)\} \\
& \text{step } (\text{Par } []) = \{(\text{Empty}, \text{Empty})\} \\
& \text{step } (\text{Assign } u \text{ } rt) = \text{step } rt \\
& \text{step } \text{Leaf } c = \{(c, \text{Empty})\} \\
& \text{step } \text{Empty} = \{(\text{Empty}, \text{Empty})\}
\end{aligned}$$

(B) Helper function step

FIGURE 2.3: Semantics of RuleTree application

$$\begin{aligned}
\text{RuleSet } a &= \mathcal{P}(\text{CR } a) \\
\text{Trace } a &= \text{Step } a \text{ (RuleSet } a \text{) (Trace } a \text{)} \\
&\quad | \text{ State } a
\end{aligned}$$

FIGURE 2.4: Definition of RuleSet and Trace.

2.4.1 RuleTree observations

The basis of the trace semantics of our multi-user rule-based problem consists of the functions $\mathcal{F}$ and $\mathcal{E}$, listed in Figure 2.5 and Figure 2.6.

$$\begin{aligned}
\mathcal{F} : \text{RuleTree } a \times a &\rightarrow \mathcal{P}(\text{RuleSet } a \times \text{RuleTree } a) \\
\mathcal{F} (\text{Seq } (rt:rts), s) &= \\
&\begin{cases} \{(\bar{R}, \text{Seq } (rt':rts)) \mid (\bar{R}, rt') \in \mathcal{F} (rt, s)\} \\ \cup \{x \mid \mathcal{E}(rt), x \in \mathcal{F} (\text{Seq } rts, s)\} & \mathcal{F} (rt, s) \neq \perp \\ \perp & \mathcal{F} (rt, s) \equiv \perp \end{cases} \\
\mathcal{F} (\text{Seq } [], s) &= \emptyset \\
\mathcal{F} (\text{Choice } (rt:rts), s) &= \mathcal{F} (rt, s) \cup \mathcal{F} (\text{Choice } rts, s) \\
\mathcal{F} (\text{Choice } [], s) &= \perp \\
\mathcal{F} (\text{Par } [rt_1, \dots, rt_n], s) &= \{(\bar{R}, \text{Par } [rt'_1, \dots, rt'_n]) \\
&\quad \mid (\bar{R}_i, rt'_i) \in (\mathcal{F}(rt_i, s) \cup \{(\emptyset, rt_i)\}) \\
&\quad, \bar{R} = \bar{R}_1 \cup \dots \cup \bar{R}_n \\
&\quad, \bar{R} \neq \emptyset \\
&\quad, \forall u_x @ r_i, u_y @ r_j \in \bar{R} : r_i(r_j \cdot s) = r_j(r_i \cdot s) \\
&\quad, \forall u_x @ r_p, u_y @ r_q \in \bar{R} : r_p \neq r_q \Rightarrow u_x \neq u_y\} \\
\mathcal{F} (\text{Par } [], s) &= \emptyset \\
\mathcal{F} (\text{Assign } u \text{ } rt, s) &= \mathcal{F} (\text{applyAssign}(rt, u), s) \\
\mathcal{F} (\text{Leaf } c, s) &= \{(\{c\}, \text{Empty})\} \\
\mathcal{F} (\text{Empty}, s) &= \emptyset
\end{aligned}$$

FIGURE 2.5: Semantics of the firsts observation $\mathcal{F}$

The function $\mathcal{F}$ (firsts) produces a set of elements of the form $(\bar{R}, rt)$, where $\bar{R}$ is a set of CR a -elements. $\bar{R}$ contains all rules that are executed at the same time. It contains at most one rule per user and all rules in this set are independent.

Function $\mathcal{E}$ (empty) checks if a RuleTree is empty. A RuleTree is considered empty if at least one of the applications of the tree does not execute any rules. For example, the empty list sequence ($\text{Seq } []$) is empty, since it holds no rules. A tree can be empty even when $\mathcal{F}$ returns a ruleset. This is the case for the RuleTree $\text{Choice } [\text{Seq } [], \text{Rule } n \text{ } e]$, for example, since when one chooses the first element, no rule is applied. But $\mathcal{F}$ returns a set containing Rule $n \text{ } e$.

$$\begin{array}{ll}
\mathcal{E} : \text{RuleTree } a \rightarrow \text{Bool} & \\
\mathcal{E} (\text{Seq } (rt:rts)) &= \mathcal{E} (rt) \wedge \mathcal{E} (\text{Seq } rts) \\
\mathcal{E} (\text{Seq } []) &= \text{True} \\
\mathcal{E} (\text{Choice } (rt:rts)) &= \mathcal{E} (rt) \vee \mathcal{E} (\text{Choice } rts) \\
\mathcal{E} (\text{Choice } []) &= \text{False} \\
\mathcal{E} (\text{Par } (rt:rts)) &= \mathcal{E} (rt) \wedge \mathcal{E} (\text{Par } rts) \\
\mathcal{E} (\text{Par } []) &= \text{True} \\
\mathcal{E} (\text{Assign } u \text{ } rt) &= \mathcal{E} (rt) \\
\mathcal{E} (\text{Empty}) &= \text{True} \\
\mathcal{E} (\text{Leaf } c) &= \text{False}
\end{array}$$
FIGURE 2.6: Semantics of the empty observation $\mathcal{E}$

Building the set of first rulesets is not trivial in a multi-user setting. This especially shows in the case of Par . This is due to the fact that parallel RuleTrees allow multiple users to execute rules at the same time.

To calculate $\mathcal{F} (\text{Par } rts, s)$, we calculate $\mathcal{F}$ for every RuleTree that is executed in parallel. Since we do not have to execute a rule from every parallel RuleTree at each step, we add the empty ruleset with the original RuleTree $((\emptyset, rt_i))$ to the set of $\mathcal{F}$. For each RuleTree rt_i in rts , we now pick one element of this $\mathcal{F}$ set that also contains the empty ruleset. Then, we put all the selected rules for each rt_i together to build the total ruleset $\bar{R}$. The remaining RuleTree is built by concatenating all rt'_i elements. These could just be the original RuleTree rt_i , if the selected element was the empty set.

Three conditions must hold for any $\bar{R}$. First, we require $\bar{R}$ to be non-empty. Second, we require every pair of elements in $\bar{R}$ to be independent, meaning that the order of application to s does not influence the resulting state. And third, we verify that there is at most one rule assigned to every user.

Function $\mathcal{F}$ relies on several auxiliary functions listed in Figure 2.7.

2.4.2 Traces of RuleTrees

Now that we have defined the firsts and empty observation, the traces function can be constructed. Fig. 2.8 lists the definition of this function.

The function traces takes a RuleTree and state, and returns the set of all possible traces. $\mathcal{F}$ is called on the RuleTree. This returns a ruleset, paired with the remaining RuleTree. These rulesets represent every possible action that can be taken. For each ruleset, a new state is calculated by applying the set to the current state. Then traces is calculated recursively to calculate the rest of the trace. When a RuleTree is empty ($\mathcal{E}(rt)$), the trace is completed, and the current state is returned.

$$\begin{aligned}
& \cdot : \text{RuleSet } a \times a \rightarrow a \\
& \bar{R} \cdot s = \\
& \left\{ \begin{array}{ll} \bar{R} \setminus \{\text{Leaf } (u @ (\text{Rule } n e))\} \cdot (e s) & \text{Leaf } (u @ (\text{Rule } n e)) \in \bar{R} \\ \bar{R} \setminus \{\text{Leaf } (\text{Rule } n e)\} \cdot (e s) & \text{Leaf } (\text{Rule } n e) \in \bar{R} \\ \bar{R} \setminus \{\text{Leaf } (\text{Cond } p c)\} \cdot (c \cdot s) & \text{Leaf } (\text{Cond } p c) \in \bar{R}, p s \\ \not\downarrow & \text{Leaf } (\text{Cond } p c) \in \bar{R}, \neg(p s) \end{array} \right. \\
& \text{(A) Semantics of ruleset application}
\end{aligned}$$

$$\not\downarrow \cup \not\downarrow = \not\downarrow \quad A \cup \not\downarrow = A \quad \not\downarrow \cup A = A \quad A \cup B = \{x \mid x \in A \vee x \in B\}$$

(B) Semantics of union

$$\begin{aligned}
& \text{applyAssign} : \text{RuleTree } a \times \text{User} \rightarrow \text{RuleTree } a \\
& \text{applyAssign}(\text{Seq } [rt_1, \dots, rt_n], u) = \text{Seq } [\text{Assign } u \ rt_1, \dots, \text{Assign } u \ rt_n] \\
& \text{applyAssign}(\text{Choice } [rt_1, \dots, rt_n], u) = \text{Choice } [\text{Assign } u \ rt_1, \dots, \text{Assign } u \ rt_n] \\
& \text{applyAssign}(\text{Par } [rt_1, \dots, rt_n], u) = \text{Par } [\text{Assign } u \ rt_1, \dots, \text{Assign } u \ rt_n] \\
& \text{applyAssign}(\text{Leaf } (\text{Cond } p c), u) = \text{Leaf } (\text{Cond } p \ (\text{Assign } u \ r)) \\
& \text{applyAssign}(\text{Assign } u_2 \ rt, u_1) = \text{Assign } u_2 \ rt \\
& \text{applyAssign}(\text{Leaf } r, u) = \text{Leaf } (u @ r) \\
& \text{applyAssign}(\text{Empty}, u) = \text{Empty}
\end{aligned}$$

(C) Definition of applyAssign

FIGURE 2.7: auxiliary definitions

This completely describes our trace semantics.

2.5 Solving algorithms

For the purpose of constructing hints, traces are of limited interest. A RuleTree includes all steps that can be taken, and therefore possibly also incorrect steps. Instead, we would like to obtain traces that end in a state that satisfies the goal the user is trying to reach.

To achieve this, we develop several solving algorithms. All algorithms return traces that may not completely apply the RuleTree, as opposed to traces, which only returns traces that have fully applied the RuleTree.

2.5.1 Breadth First Trace

The first algorithm we introduce is a breadth first trace algorithm. It performs a breadth first search, to find a state that satisfies the goal condition g . Fig. 2.9 lists its definition.

$$\begin{aligned}
& \text{traces} : \text{RuleTree } a \times a \rightarrow \mathcal{P}(\text{Trace } a) \\
& \text{traces } (rt, s) = \\
& \begin{cases} \{\text{State } s \mid \mathcal{E}(rt)\} & \\ \cup \{s \xrightarrow{\bar{R}} x \mid (\bar{R}, rt') \in \mathcal{F}(rt, s), x \in \text{traces}(rt', \bar{R} \cdot s)\} & \mathcal{F}(rt, s) \neq \perp \\ \emptyset & \mathcal{F}(rt, s) = \perp \end{cases}
\end{aligned}$$

FIGURE 2.8: Definition of the traces function.

$$\begin{aligned}
& \text{BFTrace} : \text{Goal } a \times \text{RuleTree } a \times a \rightarrow \mathcal{P}(\text{Trace } a) \\
& \text{BFTrace } (g, rt, s) \\
& \quad | g \ s \\
& \quad \mapsto \{\text{State } s\} \\
& \quad | \neg g \ s, \exists (\bar{R}, rt', s') \in \text{expand}(rt, s) : g \ s' \\
& \quad \mapsto \{s \xrightarrow{\bar{R}} \text{State } s' \mid (\bar{R}, rt', s') \in \text{expand}(rt, s), g \ s'\} \\
& \quad | \neg g \ s, \forall (\bar{R}, rt', s') \in \text{expand}(rt, s) : \neg(g \ s') \\
& \quad \mapsto \{s \xrightarrow{\bar{R}} x \mid (\bar{R}, s', rt') \in \text{expand}(rt, s), x \in \text{BFTrace}(g, rt', s')\}
\end{aligned}$$

FIGURE 2.9: BFTrace search algorithm definition

Going over the definition from top to bottom, one of three cases applies.

- If the goal is satisfied, the set containing only the current state is returned.
- If there exists one or more expansions that satisfy the goal, the traces that belong to those expansions are returned.
- If none of the expansions satisfies the goal test, BFTrace is called recursively.

2.5.2 Heuristic Trace

A possible disadvantage of the breadth first trace is that it expands all traces, and can be very slow or even infeasible, depending on the complexity of the problem. An often used solution is to perform a best first search. This method uses a heuristic function to score each expansion, and then selects the best state to further expand. If in the set of current expanded traces e there is an expansion that fulfils the goal condition, it is returned, else we recurse on the expansions that have the lowest heuristic score. The definition of our heuristic trace function is given in Fig. 2.10.

$$\begin{aligned}
\text{hTrace} &: (\text{Goal } a) \times (a \rightarrow \text{Integer}) \times \mathcal{P}(\text{RuleTree } a \times a \times \text{Trace } a) \\
&\rightarrow \mathcal{P}(\text{Trace } a) \\
\text{hTrace } (g, h, e) & \mid \exists (rt, s, x) \in e : g \ s \mapsto \{x \mid (rt, s, x) \in e, g \ s\} \\
& \mid \forall (rt, s, x) \in e : \neg g \ s \mapsto \text{hTrace } (g, h, \text{lowExp} \cup \text{high}) \\
\text{where} \\
\text{high} &= \{(rt, s, x) \mid (rt, s, x) \in e, \forall (_, s_i, _) \in e : h \ s > h \ s_i\} \\
\text{low} &= \{(rt, s, x) \mid (rt, s, x) \in e, \exists (_, s_i, _) \in e : h \ s \leq h \ s_i\} \\
\text{lowExp} &= \{(rt', s', x \xrightarrow{\bar{R}} \text{State } s') \mid (rt, s, x) \in \text{low} \\
&\quad, (\bar{R}, rt', s') \in \text{expand } (rt, s)\}
\end{aligned}$$

FIGURE 2.10: hTrace search algorithm definition

hTrace takes as argument a tuple containing the goal test g , a heuristic scoring function h and the set of current expansions e . We require h to a monotonically decreasing function, which returns a lower value as the state comes closer to the desired goal g . Initially, this set will contain only one element, namely $(rt, s, \text{Leaf } s)$, where rt is the initial RuleTree, s the initial state, and $\text{Leaf } s$ the trace that just contains the current state. If the set of current expansions contains one or more traces that lead to the goal, the algorithm returns those traces. If none of the expansions lead to the goal, the expansions are scored using the scoring function h , and divided into two sets, one containing the highest scoring expansions, and one containing the others. The highest scoring expansions are then expanded. hTrace is called recursively on the union of the expanded traces and the low scoring traces.

2.6 Implementation

Our framework has been implemented in Haskell. Haskell is a purely functional programming language. It has a static type system and lazy evaluation. While this helps with the implementation, it is not crucial to the realisation of the system.

Listing 2.1 lists the types of the functions that correspond to the functions described in Sections 2.3 to 2.5. The full implementation can be found online ¹.

We have also implemented two examples that use the framework to generate hints: Tic Tac Toe and a command and control system. Both examples are included in the full implementation available online. We discuss

¹<https://github.com/niconaus/rule-tree-semantics>

```

firsts :: Eq a => RuleTree a -> a -> Maybe [(RuleSet a, RuleTree a)]
empty  ::      RuleTree a -> Bool
expand :: Eq a => RuleTree a -> a -> Maybe [(RuleSet a, RuleTree a)]
traces :: Eq a => RuleTree a -> a -> [Trace a]

BFTrace :: Eq a => (Goal a) -> [(RuleTree a, a, [(a, RuleSet a)])]
        -> [Trace a]
heuristicTrace :: Eq a => (Goal a) -> (a -> Int)
        -> [(RuleTree a, a, [(a, RuleSet a)])] -> [Trace a]

```

LISTING 2.1: Type signatures of the framework implementation

the command and control example in the following section. Chapter 3 lists several other examples, and lists applications of the framework beyond the workflow domain.

2.6.1 Properties of the traces function

To validate our definition of $\mathcal{F}$, $\mathcal{E}$, `expand` and `traces`, we want to show them to be correct.

We do this by verifying the `traces` function to be sound and correct with respect to the `RuleTree` application semantics.

We consider `traces` to be sound if, for any `RuleTree` rt and initial state s , there exists an end state in the result of $rt \cdot s$ that is equal to the end state reached by every trace in `traces`(rt, s).

We consider `traces` to be complete if for all elements in the set of end states from the application of the `RuleTree`, there exists an element from `traces`, such that the end state of this trace is equal to the element of the end state set. Instead of showing soundness and completeness separately, we verify Conjecture 2.6.1, from which we can deduce the two.

Conjecture 2.6.1 (Correctness of traces)

For all `RuleTrees` rt and states s we have:

$$\{s_n \mid s \xrightarrow{\bar{R}_1} \dots \xrightarrow{\bar{R}_n} s_n \in \text{traces}(rt, s)\} = rt \cdot s.$$

We verify that our implementation works correctly by testing the correctness properties as formulated in Conjecture 2.6.1, using QuickCheck (Claessen & Hughes, 2000). QuickCheck generates random test cases for properties, based on the type signature of the input of a property. The translation of this Conjecture to Haskell is listed in Listing 2.2.

```
rtEquality :: RuleTree [Int] -> [Int] -> Property
rtEquality rt s = (fromList (traceS rt s)) == (fromList (appS rt s))
```

LISTING 2.2: Correctness property Conjecture 2.6.1 expressed in Haskell

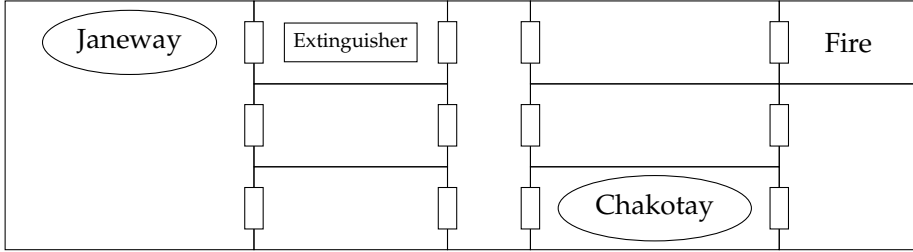


FIGURE 2.11: Rendering of an example initial state for the simplified Command & Control system with two workers: Janeway and Chakotay

2.6.2 Command & Control system

We take a look at a Command & Control application that was developed in cooperation with the Netherlands Royal Navy (Stutterheim et al., 2016). The goal of this application is to model workflows on board a navy ship. This includes workers, sensors, mission goals, resources and systems. Tasks can be assigned to users working on the vessel, and sensors are used to monitor the current situation.

For the sake of this example, we use a simplified version of the complete ship application. Workers can walk around the ship. When a fire breaks out, the workers have to walk to an extinguisher, pick it up, walk to the fire, and put it out. A visual representation of this example is shown in Fig. 2.11.

The code below shows how we describe the problem in our DSL. Only the most important definitions are given. For the goal and heuristic functions, only the type signature is given here. The complete definitions can be found online².

```
data SimulationState = SimulationState [[Room]] (M.Map User Agent)

data User = User String
data Agent = Agent RoomNumber -- Current position
              Inventory
              User              -- User that controls Agent

data Room = Room RoomNumber
              (Int,Int)      -- Room coordinates
              [Exit]         -- Rooms it has doors to
              Inventory
```

²<https://github.com/niconaus/rule-tree-semantics>


```

RoomState
Int          -- Room depth

data Exit = ENorth RoomNumber
          | EEast RoomNumber
          | ESouth RoomNumber
          | EWest RoomNumber

data Inventory = NoItem | Extinguisher
data RoomState = Normal | Fire

shipTree :: RuleTree SimulationState
shipTree = Parallel (map (\usr -> Assign usr (shipSimulation usr))
                      [Janeway, Chakotay])

shipSimulation :: User -> RuleTree SimulationState
shipSimulation usr
= times 10
  (Choice
   [ Leaf (Condition (canPickup usr) (pickUp usr))
   , Leaf (Condition (canExtinguish usr)
   , Choice (map (\x -> (Leaf
                        (Condition (canMove usr)
                        (Rule (show x) (applyMove usr x))))
                [1..10]))])

shipNotOnFire :: Goal SimulationState
shipHeuristic :: SimulationState -> Int

solveShip = heuristicTrace shipNotOnFire [(shipTree, shipState, [])]

```

The first line models the state, and `shipTree` expresses the `RuleTree`, with the help of `shipSimulation`.

Assuming the system itself is also implemented in Haskell, the existing code from the implementation can be used when defining the `RuleTree`. Functions like `pickUp`, `canExtinguish` and `applyMove` can be the exact same code as the system implementation.

`shipNotOnFire` is the goal condition, and `shipHeuristic` is the heuristic used to score each state. To solve this problem, we plug these functions into the generic `heuristicTrace` algorithm, together with the `RuleTree` and a state, as shown on the last line. When we execute `solveShip`, we get back a trace that will lead the workers on the ship to the quickest way to extinguish all fires, if possible. If there is only a single fire, instructions for only one user will be generated. If there are multiple fires, both workers will perform actions at the same time, as described by the `ruleTree`.

This example clearly shows the advantage of our system: a programmer only needs to define the problem by describing it as a `ruleTree`, possibly reusing existing code, come up with a goal function and a heuristic, and then gets an adaptive, multi-user solver for free.

2.7 Conclusions

In this chapter, we have demonstrated how to construct a complete and sound framework for calculating hints for multi-user workflow systems. By means of a DSL, we are able to describe problems in a uniform way, and make them tractable to generic solving algorithms. These algorithms produce traces that lead to the goal of the user. Besides a formal system, we have also presented a practical implementation. We have implemented two examples, one of which we have described in this chapter.

2.8 Related work

2.8.1 Rule-based problem modelling

We follow in a long tradition of creating (domain-specific) languages that allow programmers to model rule-based problems, such as planning problems. Some of the early languages written for this purpose are STRIPS (Fikes & Nilsson, 1971), PLANNER (Hewitt, 1969) and SITPLAN (Galagan, 1979). Most of these are based on the same principles as our approach, namely to describe state, operator set and goal test. For example, a STRIPS problem is defined as $\langle P, O, I, G \rangle$, where P is the set of states the problem can be in, O the set of operators, I is the initial state, and G the goal state (Bylander, 1994).

A more recent language is PDDL (McDermott et al., 1998). Version one of the language, from 1998, consists of a domain description, action set, goal description and effects. Again, these ideas coincide with our notion of a problem formalization. The PDDL standard has been updated several times (Kovacs, 2011), and there are many variants currently in use. These variants include MA-PDDL (Kovacs, 2012), which can deal with multiple agents, and PPDDL (Younes & Littman, 2004), which supports probabilistic effects.

The language we present is different from all of the aforementioned languages in several ways. Our language is a DSL, embedded in Haskell. This means that the programmer can use the full power of Haskell when constructing the problem description in our DSL. The languages mentioned

above are not embedded in any language and therefore the programmer is limited to the syntax of the DSL in constructing the problem description. Another big difference is the fact that in all of the other languages mentioned, except PDDL, the state-space is finite. For example, in SITPLAN, part of the problem description is a finite set of possible situations, and in STRIPS, the set of states is defined as finite a set of conditions that can be either true or false. In our DSL, we do not limit the set of possible states. This allows us to describe many more problems in our DSL, but at the same time makes solving them harder.

The second part of our approach is to solve the problem described in our DSL. When comparing to other approaches, both SITPLAN and PDDL rely on general solvers, just like our approach. In fact, PDDL was initially designed as a uniform language to compare different planning algorithms in the AIPS-98 competition (McDermott et al., 1998). STRIPS and PLANNER however, do include a specific solving algorithm.

For each of the frameworks that we discussed in this section, there has been some research on generically solving problems. The Ideas framework includes a set of feedback services to generate hints for the user. For example, the `basic.allfirsts` service generates all steps that can be taken at a certain point in the exercise (Heeren & Jeuring, 2014). For the iTasks framework, a system was developed to inspect current executions by using dynamic blueprints of tasks (Stutterheim, Achten, & Plasmeijer, 2015). It can give additional insight in the current and future states, but does not act as a hint-system and does not take a goal into account.

2.8.2 Workflow Analysis

There has been done some work on analysing instances of workflow systems. Basu and Blanning present metagraph (Basu & Blanning, 2000) to describe workflows so that they can be better evaluated. Other approaches apply workflow mining to evaluate implementations (Aalst, 2011). Stutterheim et al. (Stutterheim, Plasmeijer, & Achten, 2014) present a system for generating visualisations from the source code of workflow systems implemented in the iTasks workflow framework. Their system Tonic also features dynamic inspection and limited path prediction. These approaches do not use their analyses to assist the end-user. Instead they focus on workflow and business optimisation from the system design perspective.

Research has also been done on systems that help end users in making choices. These decision support systems usually leverage some artificial intelligence approach like probabilistic reasoning (Pearl, 1989) or planning (Kaelbling, Littman, & Cassandra, 1998). These are all solutions that

are custom made for a specific workflow system instance.

To our knowledge, we are the first to describe a workflow solving system that works generically on a broad range of problems structured in a workflow.

2.8.3 Decision Support Systems

A Decision Support System is defined as a system which models a certain domain and then assists the user in making choices by using analysis techniques (Shim et al., 2002).

There exists a great variety in both domains where DSSs are applied, as well as their implementation. Clinical DSSs support making decisions about the treatment of individual patients (Berner & La Lande, 2007). There are agricultural DSSs aimed to improve land use, planning and management of soil (Rosa, Mayol, Díaz-Pereira, Fernandez, & Rosa Jr., 2004). The biggest area of application is management and business (Turban, 1988). There, DSSs help managers make the right choices faster, better allocate resources or identify trends.

The basic design of a DSS consists of some representation of the domain, a reasoning engine and a way to communicate with the user.

Using a DSS has many advantages (Power, 2002). It improves the productivity of individuals, improves the quality of decisions and the speed with which they are made. Organisational control is improved, as well as communication between workers.

Employing a DSS comes with several challenges. First of all, there is a large financial risk involved, since it requires a significant investment (Power, 2002). The model that is used in the DSS limits the applicability of the system. When the domain or the problem changes, the model needs to be updated as well. Social issues may come up as well, workers may resist the change that comes with a DSS.

2.8.4 Electronic Performance Support Systems

Electronic Performance Support Systems (EPSS) focus on workers or individuals that have to achieve a certain goal or complete a task, but who do not yet have sufficient knowledge or are not sufficiently skilled yet. They facilitate on the job training by providing the user with just-in-time information on the task that they are working on (Schaik et al., 2002).

An EPSS is typically composed of a user interface, giving access to generic tools like documentation and help systems, and application specific support tools such as tutorials (Barker & Banerji, 1995). Usually, the EPSS is

geared towards the specific domain it is being used in, a certain business setting for example.

An EPSS can provide workers with just in time information on how to perform certain tasks. It cannot however assist hem in making decisions based on the precise situation that they are in. Only general documentation, help and guidelines can be offered.

The aim of our next-step hint system is not necessarily to provide training to workers, but to assist them with a specific goal and situation.

Chapter 3

Generating next-step hints for tasks, puzzles and exercises

In Chapter 2, we have developed a multi-user generic feedback framework for rule-based problems. In this chapter, we want to demonstrate that our approach can be applied to different problems. These problems come from a broad range of domains, namely computer games, intelligent tutoring systems (ITS) and workflow systems. By targeting a framework from each of these domains, our solution is immediately applicable to many problems formulated in one of these frameworks. We show that it is indeed feasible to encode these problems, which are rather different in nature, in our feedback framework, and that it is possible to calculate feedback. In all three domains, providing feedback to end users is essential. In an ITS, feedback is used by students to learn how to solve exercises, providing feedback to players of computer games will keep them interested, and by providing feedback to users of a workflow system, the quality of the choices they make will improve.

3.1 Introduction

This chapter demonstrates our goal of providing help to people using a rule-based problem-solving system by giving examples from three different domains: intelligent tutoring systems (ITS), computer games and workflows. We do this by targeting frameworks from these domains: the Ideas framework (Heeren et al., 2010), PuzzleScript (Lavelle, 2016) and the iTasks system (Plasmeijer et al., 2012). Each of these frameworks can describe a variety of problems. We briefly introduce each framework, show an example problem described in that framework, and then show how to solve the problem by using the generic framework from the previous chapter.

3.2 Ideas

The Ideas framework is used to develop services to support users when stepwise solving exercises in an intelligent tutoring system for a domain like mathematics or logic. It is a general framework used to construct the expert knowledge of an intelligent tutoring system (ITS). The framework has been applied in the domains of mathematics (Heeren et al., 2010), programming (Gerdes, Jeuring, & Heeren, 2012), and communication skills (Jeuring et al., 2015).

```
dnfStrategy = label "Constants"  (repeat (topDown constants))
              <*> label "Definitions" (repeat (bottomUp definitions))
              <*> label "Negations"   (repeat (topDown negations))
              <*> label "Distribution" (repeat (somewhere distribution))
```

FIGURE 3.1: A problem-solving strategy in Ideas

The central component of the expert knowledge for an ITS is expressed as a so-called *strategy* in Ideas. For example, Figure 3.1 gives part of a strategy for the problem of rewriting a logic expression to disjunctive normal form (for the complete strategy see Heeren et al. (Heeren et al., 2010)). The framework offers various services based on this strategy, among which a service that diagnoses a step from a student, and a service that gives a next step to solve a problem. The student receives a logic expression, and stepwise rewrites this expression to disjunctive normal form using services based on the above strategy. At each step, the student can request a hint, which will look something like “Eliminate constants” or “Eliminate implications”, or ask for feedback on her current expression. If no rules can be applied any more, the expression is in normal form.

The `dnfStrategy` describes a rule-based process that solves the problem of converting an expression to disjunctive normal form. It is expressed in terms of combinators like `<*>` (sequence), `repeat` and `somewhere`, and other sub-strategies. Additionally, a `label` combinator is available, to label sub-strategies with a name.

3.2.1 Using RuleTree to describe Ideas strategies

We have taken two examples, with different domains and problems, implemented in Ideas: calculating the disjunctive normal form of a logic expression (see Fig. 3.2), and reducing a matrix to echelon form (see Fig. 3.3). By implementing these strategies as RuleTrees, we demonstrate that our framework can indeed be used in an ITS.

Disjunctive Normal Form

Figure 3.2 gives the description of the disjunctive normal form exercise in our DSL. This is almost a direct translation from the Ideas strategy given above in Fig. 3.1.

```
dnf :: RuleTree Expr
dnf = Seq [ repeat (Leaf constantsR), repeat (Leaf definitionsR)
          , repeat (Leaf negationsR), repeat (Leaf distributionR)]

repeat :: RuleTree a -> RuleTree a
repeat rt =
  Choice [ Condition (not.(empty rt)) (Seq [ rt, repeat rt])
        , Condition (empty rt) Empty]
```

FIGURE 3.2: DNF exercise in our DSL

Fig. 3.2 shows the RuleTree for the DNF strategy. The state is modelled by expressions of type `Expr`, representing the current state of the exercise of the student. To encode it compactly, an additional function is used called `repeat`. This function checks if the RuleTree is non empty. If so, the RuleTree can be applied, after which `repeat rt` is called again. If not, the Empty RuleTree is returned and this will end the recursion. This means that the RuleTree that `repeat` is applied to, should have an effect on the condition, otherwise this recursion will never terminate.

The RuleTree is all that is required to build the hint-function. Since all steps offered by the RuleTree are on a path to the goal, we can just return them by using the RuleTree algorithm.

The hint function below takes an expression of type `Expr` representing the current state, and returns steps that can be taken at this point. If no steps are returned, the exercise is solved.

```
hint :: Expr -> [RuleSet Expr]
hint e = map fst (maybeToList (firsts dnf e))
```

The RuleTree `dnf` potentially does not terminate, since it relies on `repeat`. This is not a problem however, since a hint is produced by simply returning the first steps that are available.

Gaussian Elimination

Our second example is in the domain of linear algebra. The exercise at hand is to reduce a matrix to echelon form, using Gaussian elimination.


```
toReducedEchelon = label "Gaussian elimination"
                    (forwardPass <*> backwardPass)

forwardPass = label "Forward pass" (
  repeat (
    label "Find j-th column"    ruleFindColumnJ
  <*> label "Exchange rows"      (try ruleExchangeNonZero)
  <*> label "Scale row"          (try ruleScaleToOne)
  <*> label "Zeros in j-th column" (repeat ruleZerosFP)
  <*> label "Cover up top row"   ruleCoverRow))

backwardPass = label "Backward pass" (
  repeat ( label "Uncover row" ruleUncoverRow
    <*> label "Sweep" (repeat ruleZerosBP)))
```

FIGURE 3.3: Gaussian elimination strategy in Ideas

Figure 3.3 lists the strategy of Gaussian elimination that is used in the Ideas framework. It describes what steps must be applied to a matrix to transform it to the reduced echelon form, by means of Gaussian elimination.

The forward pass is applied to the matrix as often as possible. When this procedure no longer applies, the backwards pass is applied exhaustively. If no rules from either two phases apply, the matrix has been reduced. We leave out the exact details of what each rule in the passes does, they are available elsewhere (Heeren et al., 2010).

To describe the strategy in a similar way in our DSL, we need to introduce one new function to deal with try. Figure 3.4 lists the complete description of Gaussian elimination in our DSL, together with our try definition.

try takes a rule, and then checks if the rule has an effect if it is applied by means of the canApply function. If it has an effect, the rule can be selected. Otherwise the empty RuleTree is returned.

Since we are again dealing with a RuleTree where all the offered steps are on a path to the goal, no goal test function is needed to build the hint-function.

```
hint :: Expr -> [Name]
hint e = map fst (maybeToList (firsts toReducedEchelon e))
```

As with the DNF example, applying firsts to the RuleTree and current expression instantiates the hint-function.

```

toReducedEchelon :: RuleTree Expr
toReducedEchelon = Seq [forwardPassRT, backwardsPassRT]

forwardPassRT = repeat Seq [Leaf ruleFindColumnJ
                           ,tryRule ruleExchangeNonZero
                           ,tryRule ruleScaleToOne
                           ,repeat (Leaf ruleZerosFP)
                           ,Leaf ruleCoverRow]

backwardPassRT = repeat Seq [Leaf ruleUncoverRow
                           , repeat (Leaf ruleZerosBP)]

try :: Rule a -> RuleTree a
try rule = Choice [ Condition (canApply rule) (Leaf rule)
                  , Condition (not.(canApply rule)) Empty]

canApply :: Rule a -> a -> Bool
canApply (Rule n e) s = (e s) /= s

```

FIGURE 3.4: Gaussian elimination exercise in our DSL

3.3 PuzzleScript

PuzzleScript is an open source HTML5 Puzzle Game Engine (Lavelle, 2016). It is a simple scripting language for specifying puzzle games. Its central component is a DSL for describing a game. PuzzleScript compiles a puzzle described in this DSL into an HTML5 puzzle game. Using the DSL, the game programmer describes a puzzle as a list of objects, rules that define the behaviour of the game, a win condition, collision information, and one or more levels.

The hello-world example for PuzzleScript is given in Figure 3.5. It describes a simple crate-pusher game, also called Sokoban. Objects are: background, walls, crates, the player and the targets for the crates. There is a single rule that states if a player moves into a crate, the crate moves with the player. Objects appearing on the same line in the collision layers are not allowed to pass through each other. The winning condition is reached when all targets have a crate on them. Finally, a start-level is specified under LEVELS.

In a difficult game, we want to offer next-step hints to the player on how to proceed. Based on the state of the game, the RULES, COLLISIONLAYERS and WINCONDITIONS, an algorithm can calculate a hint for a user (Lim & Harrell, 2014). This same information can also be used to check if a game can still be solved in the current state. For example, the game cannot be solved any

```

=====
OBJECTS      LEGEND
=====
Background   . = Background
Green        # = Wall
             P = Player
Target       * = Crate
DarkBlue     @ = Crate and Target
             O = Target

Wall
Brown
Player
Blue
Crate
Orange

=====
RULES
=====
[ > Player | Crate ]
-> [ > Player | > Crate ]

=====
WINCONDITIONS
=====
All Crate on Target

=====
LEVELS
=====
# # # # # # # # # #
# . . . . . . . #
# . . . . . @ . #
# . P . * . O . #
# . . . . . . . #
# . . . . . . . #
# # # # # # # # # #

=====
COLLISIONLAYERS
=====
Background
Target
Player, Wall, Crate

```

FIGURE 3.5: Partial definition of the hello-world example of PuzzleScript

more if a crate gets stuck in a corner.

3.3.1 Solving Sokoban

Figure 3.6 lists the RuleTree for Sokoban. We first define GameState which models our state. It contains the LevelState, as well as the position of the player pX, pY . sokoban defines the RuleTree. In sequence, it offers choice from one of the four moves, and then recurses. All moves are conditional, they can only be chosen if they can actually be applied. We only supply the types of the functions validMove and applyMove.

On first attempt, we take the breadth first search algorithm BFTrace, and use it to construct our hint-function. For trivial levels, this suffices, but once we try to generate hints for levels where a solution consists of 15 moves, we have to explore $3^{15} \approx 1.4 \times 10^7$ states, assuming that there are on average three valid moves per state.

```

data GameState = {lvl :: LevelState , pX :: Int , pY :: Int}
type LevelState = [[[GameObject]]]
data GameObject = Target | Wall | Player | Crate

sokoban :: RuleTree GameState
sokoban = repeat (Choice [ move "Move Left" LeftMove
                          , move "Move Right" RightMove
                          , move "Move Up" UpMove
                          , move "Move Down" DownMove])

move :: String -> GameMove -> RuleTree GameState
move name step = Condition (validMove step)
                        (Leaf (Rule name (applyMove step)))

sokobanGoal :: GameState -> Bool

validMove :: GameMove GameState -> Bool
applyMove :: GameMove GameState -> GameState

```

FIGURE 3.6: Sokoban in our DSL

Breadth first search clearly will not work. We have to come up with something a bit more informed. Literature on Sokoban (Junghanns & Schaeffer, 1997) points to heuristics and search space pruning to help us order and restrict the search space, and construct a hint-function. Lim and Harrell have generalized these Sokoban heuristics to apply to most PuzzleScript games (Lim & Harrell, 2014).

We implement a simple deadlock pruning filter to improve performance. A simple deadlock occurs when a crate is in an unsafe position, from where it will never reach a target. Removing these states reduces the search space.

Implementing heuristics for Sokoban, like the mentioned work suggests, can be quite involved. This is beyond the scope of this chapter, but could be implemented using the `heuristicHint` algorithm.

To perform simple deadlock detection, we first build a list of positions from where we will never reach a target. To do this, we find all the corners in the game. After locating the corners, we generate a list of all horizontal and vertical paths from corner to corner. Paths that are not along a wall, or that have walls or targets on them, are removed. The cells on the remaining paths, together with the corners, form the list of unsafe positions. To determine if a state has a deadlock, we simply inspect all unsafe positions. If a state has a crate on an unsafe position, it is removed and thus not further expanded.

Below, the hint function is implemented. For the simple deadlock pruning function, we only provide the type.

```
noDeadlock :: GameState -> Bool

BFTFilter :: (a -> Bool) -> (a -> Bool) -> RuleTree a -> a
          -> [RuleSet a]

hint :: GameState -> Name
hint state = map fst (maybeToList (
    BFTFilter (\ (b,_,_) -> noDeadlock b )
              sokobanGoal
              sokoban
              state))
```

The function `BFTFilter` is a variant of the `BFTrace` function that takes as an additional argument, a pruning function from state to boolean. Only its type is provided.

3.4 iTasks

`iTasks` (Plasmeijer et al., 2012) supports task-oriented programming in the pure functional programming language Clean (Plasmeijer, van Eekelen, & van Groningen, 2002). It allows for rapid workflow program development, by using the concept of task as an abstraction. Clean is very similar to Haskell, with a few exceptions. A data declaration starts with `::`, types of function arguments are not separated by a function arrow (`->`) but by a space, and class contexts are written at the end of a type, starting with a `||`.

An `iTasks` program is composed out of base tasks, task combinators, and standard Clean functions. A task is a monadic structure. Its evaluation is driven by events and handling an event potentially changes a shared state. Tasks can be combined using combinators. The most common combinators are `>>=` (sequence), `>>*` (step), `-||-` (parallel) and `-&&-` (choice). The step combinator can be seen as a combination of sequence and choice. It takes a task and attaches a list of actions to it, from which the user can choose. The chosen action receives a result value from the first task. The action, which is of type `TaskStep`, is a regular task combined with an action to trigger it, and a condition that must hold for the action to be available.

3.4.1 Solving a sliding puzzle

To demonstrate and experiment with `iTasks`, we implement a simple sliding puzzle (also called `n-puzzle`).

Figure 3.7 gives the (partial) source code of the `iTasks` program that we constructed. In this puzzle, the player arranges all tiles in order, by using the hole to slide the tiles over the board, as shown in Figure 3.8.

```

:: GameState = { board :: [Int], dim :: Int, hole :: Int}
:: Dir = North | East | South | West
boardStore :: Shared GameState

slidePuzzle :: Task GameState
slidePuzzle =
  viewSharedInformation "Sliding Puzzle" [ViewWith viewBoard] boardStore
  >>* map (\dir -> OnAction (Action (toName dir) [])
        (ifValue (checkStep dir)
          (\st -> set (applyStep dir st)
            boardStore >>| slidePuzzle)))
    [North, East, South, West]

viewBoard :: GameState -> HtmlTag
checkStep :: Dir GameState -> Bool
applyStep :: Dir GameState -> GameState

```

FIGURE 3.7: Sliding puzzle program written in iTasks

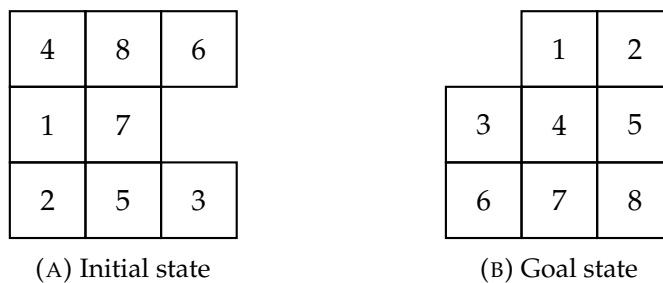


FIGURE 3.8: Instance of a block sliding puzzle, of dimension 3 x 3

```

slidePuzzle :: RuleTree GameState
slidePuzzle = repeat Choice [ slide "Move up" North
                             , slide "Move down" South
                             , slide "Move left" West
                             , slide "Move right" East]

slide :: String Dir -> RuleTree GameState
slide name direction = Condition (checkStep direction)
                                (Rule name (applyStep direction))

goalTest :: GameState -> Bool
goalTest {board, dim} = [0..((dim*dim)-1)] == board

```

FIGURE 3.9: Sliding puzzle in our DSL

The record type `GameState` holds the board configuration, the dimension of the puzzle, and the position of the hole. `Dir` defines the kind of moves a player can perform and `slidePuzzle` implements the puzzle.

`slidePuzzle` uses the standard task for viewing information to display the current state. Then, it uses the step combinator `>>*` to combine the viewing task with the tasks offering the possible options for moving the hole. We use a `map` to generate the four options a player can choose from.

The goal of the puzzle is to move all tiles in positions so that they appear in order, as shown in Figure 3.8b. We now want to add hints to the `iTasks` program that indicate to the user which step to take next in order to solve the puzzle. If a player gets stuck, she can ask for help.

Figure 3.9 lists the `RuleTree` and `goalTest` for the sliding puzzle. The functions `checkStep` and `applyStep` are the same Clean functions used by the `iTasks` implementation. The only additional function needed is the `goalTest`, that compares the current board to the solution-state.

The `n-puzzle` problem is too complex to apply a brute force algorithm. An 8-puzzle for example has an average branching factor of 2.67 (Luger, 2005), and an average solution length of 21.97 (Reinefeld, 1993). We can calculate that we have to visit $2.67^{21.97} \approx 3.39 \times 10^8$ states on average before a solution is found using brute force search.

This calls for a more informed algorithm. Russell and Norvig propose two different heuristics for the `n-puzzle` problem (Russell & Norvig, 2010). The first, h_1 , is the number of tiles out of place. The second heuristic h_2 is the sum of the (Manhattan) distances of the tiles from their goal positions. For our purposes, just using the first heuristic h_1 already suffices. With help of h_1 , we can construct the following hint function.

```
hint :: GameState -> [Trace a]
```

```

hint = heuristicTrace goalTest h1 [(slidePuzzle, state,[])]

h1 :: GameState -> Int
h1 {board, dim} = boardDiff board [0..((dim*dim)-1)]
  where
    boardDiff [] [] = 0
    boardDiff [x:xs] [y:ys] | x /= y = 1 + boardDiff xs ys
                             | True  = boardDiff xs ys

```

The function `h1` works as follows. As an argument, it receives the current `GameState` which contains the board and its dimension (`dim`). It then calls the function `boardDiff` which calculates the distance between two boards. As arguments, the current board and the solution (`[0..((dim*dim)-1)]`) are provided.

To build the hint function, we use the heuristic trace function instead of breadth first search. This expands the state space in an ordered way.

We can now run the hints function as an `iTasks` program in parallel with the original program, to provide hints to end users in the same view. With this solution, we can calculate hints for each possible state of a game with a dimension of three. Solving bigger games may require implementing the additional heuristic h_2 .

3.5 Conclusion

In this chapter, we have demonstrated that the multi-user generic feedback framework from Chapter 2 can be applied to different problems from a broad range of domains. For each of the problems, we have defined a hints function that calculates next-step hints for end users. The problems discussed in this chapter come from three different frameworks, each from a different domain. Below, we state our conclusions per framework.

3.5.1 Ideas

By implementing two example problems from the Ideas framework, we have shown that our framework can function as an alternative implementation of the Ideas framework. The strategies can be directly translated into our DSL. Implementing the auxiliary functions like `repeat` and `try`, defined by Ideas, is quite simple. All other ideas combinator functions can be implemented as well. Once the strategies have been defined, it only takes a single line of code to compute next-step hints.

We are now able to do everything the ideas framework can do, and more. Ideas calculates hints by applying the strategy that is defined to solve the

exercise. Our framework however can also take into account the specific goal the learner wants to achieve. For example, she wants to arrive at a certain specific solution, when multiple solutions are possible.

3.5.2 PuzzleScript

We have implemented one example problem from the open source Puzzle Game Engine PuzzleScript in our framework. For this game, Sokoban, we are able to compute next-step hints for rather complex problems, by filtering out configurations that are stuck. By implementing this example, we have shown that we are not only capable to deal with exercises and workflows, but can also accommodate puzzles and games.

3.5.3 iTasks

In Chapter 2, we have already demonstrated that we are able to encode workflow problems in our framework. We further substantiate this claim by implementing a second example, the n-puzzle, in our framework.

Part II

Task-oriented programming & automatic hint generation

Chapter 4

An example-based introduction to task-oriented programming

Software that models how people work is omnipresent in today's society. Current languages and frameworks often focus on usability by non-programmers, sacrificing flexibility and high-level abstraction. Task-oriented programming (TOP) is a programming paradigm that aims to provide the desired level of abstraction while still being expressive enough to describe real world collaboration. It prescribes a declarative programming style to specify multi-user workflows. Workflows can be higher-order. They communicate through typed values on a local and global level. Such specifications can be turned into interactive applications for different platforms, supporting collaboration during execution. This chapter describes the TOP paradigm by example. We will use this description to develop a formal TOP semantics, symbolic execution to verify properties, and a next-step hint system for workflow systems over the coming chapters.

4.1 Introduction

Many applications these days are developed to support workflows in institutions and businesses. Take for example expense declarations, order processing, and emergency management. Some of these workflows occur on the boundary between organisations and customers, like flight bookings or tax returns. What they all have in common is that they need to interact with different people (medical staff, tax officers, customers, etc.) and they use information from multiple sources (input forms, databases, sensors, etc.).

4.1.1 Task-oriented programming

Task-oriented programming (TOP) is a programming paradigm that targets the sweet spot between faithful modelling workflows and rapid prototyping

of multi-user web applications supporting these workflows (Plasmeijer et al., 2012). TOP focusses on modelling collaboration patterns. This gives rise to a user's need to interact and share information. Next to that, TOP automatically provides solutions to common development jobs like designing GUIs, connecting to databases, and client-server communication.

Therefore, a language that supports TOP should choose the right level of abstraction to support two things. Firstly, it should provide primitive building blocks that are useful for high-level descriptions of how users collaborate, with each other and with machines. These building blocks are: *editors*, *composition*, and *shared data*. Secondly, it should be able to generate applications, including graphical user interfaces, from workflows modelled with said building blocks.

Users can work together in a number of ways, and this is reflected in TOP by task compositions. There is *sequential* composition, *parallel* composition, and *choice*. Users need to communicate in order to engage in these forms of collaboration. This is reflected in TOP by three kinds of communication mechanisms. There is data flow *alongside* control flow, where the result of a task is passed onto the next. There is data flow *across* control flow, where information is shared between multiple tasks. Finally, there is communication with the *outside* world, where information is entered into the system via input events and output is returned via observations. The end points where the outside world interacts with TOP applications are called editors. In generated applications, editors can take many forms, like input fields, selection boxes, or map widgets.

4.1.2 Implementations of TOP

Currently, we know of two frameworks that implement TOP: iTasks and mTasks. iTasks is an implementation of TOP, in the form of a shallowly embedded domain-specific language in the lazy functional programming language Clean. It is a library that provides editors, monadic combinators, and shared data sources. iTasks uses the generic programming facilities of Clean to derive rich client and server applications from a single source. It has been used to model an incident management tool for the Dutch coast guard (Lijnse et al., 2012). Also, it has been used to prototype ideas for Command and Control systems (Kool, 2017; Stutterheim, 2017), and in a case study for the Dutch tax authority (Stutterheim et al., 2017).

mTasks is a subset of iTasks, focusing on IOT devices and deployment on micro controllers. It has been used to control home thermostats and other home automation applications (P. Koopman et al., 2018).

4.1.3 Challenges

Both iTasks and mTasks have been designed for developing real-world applications. They are constantly being extended and improved with this goal in mind. The different variations of task combinators and the details that come with real-world requirements, make it hard to see what the essence of TOP is.

In this chapter, we want to take a step back and look at the essence of TOP. We will use this description to develop a next-step hint system for workflow systems over the coming chapters. In Chapter 5, we present a formal TOP semantics. This paves the way for formal reasoning about TOP software. In Chapter 6, we will develop a symbolic execution framework based on the formal semantics that allows us to prove properties over TOP programs. The symbolic execution framework also powers the automatic next-step hint generation system presented in Chapter 7.

4.2 TOP by example

This section gives an overview of the abilities of tasks in the task-oriented programming paradigm.

4.2.1 Tasks model collaboration

The central objective of TOP is to *coordinate collaboration*. The basic building blocks of TOP for expressing collaboration are task combinators. They express ways in which people can work together. Tasks can be executed after each other, at the same time, or conditionally. This motivates the combinators step, parallel, and choice.

Example 4.2.1 (Breakfast)

The following program shows the different collaboration operators in the setting of making breakfast. Users have a choice ($\diamond$) whether they want tea or coffee. They always get scrambled vegetables. The drink and the food are prepared in parallel ($\bowtie$). When both the drink and the food are prepared, users can step ($\triangleright$) to eating the result.

```
let mkBrkfst : TASK Drink  $\rightarrow$  TASK Food  $\rightarrow$  TASK  $\langle$ Drink,Food $\rangle$ 
  =  $\lambda$ mkDrink.  $\lambda$ mkFood. mkDrink  $\bowtie$  mkFood in
    mkBrkfst (mkTea  $\diamond$  mkCoffee) mkVeganScramble
     $\triangleright$  enjoyMorning
```

The way the combinators are defined matches real life closely. When we want to have breakfast, we have to complete several other tasks first before we can do so. We decide what we want to have and then prepare it. We can prepare the different items we have for breakfast in parallel, but not at the same time. For example, it is impossible to make scrambled vegetables, and put on the kettle for tea simultaneously. Instead, what is meant by parallel is that the order in which we do tasks and the smaller tasks that they are composed of, does not matter. Then finally, only when every item we want to have for breakfast is ready, can we sit down and enjoy it.

4.2.2 Tasks are reusable

There are three ways in which tasks are modular. First, larger tasks are composed of smaller ones. Second, tasks are first-class, they can be arguments and results of functions. Third, tasks can be result values of other tasks. These aspects make it possible for programmers to model custom collaboration patterns. Example 4.2.1 demonstrates how tasks can be parameterised by other tasks: `mkBrkfst` is a collaboration pattern that always works the same way, regardless of which food and drink are being prepared.

4.2.3 Tasks are driven by user input

Input events drive evaluation of tasks. The application of a valid event to the current task, results in a new task. This is how TOP communicates with the environment. Inputs are synchronous, which means the order of execution is completely determined by the order of the inputs.

In TOP, *editors* are the basic method of communication with the environment. Editors are modelled after input widgets from graphical user interfaces. There are different editors, denoted by different box symbols. Take for example an editor holding the integer seven: $\square 7$. Such an editor reacts to change events, for example the values 42 or 37, which are of the same type.

The sole purpose of editors is to interact with users by remembering the last value that has been sent to them. There are no output events. As values of editors can be observed, for example by a user interface, editors facilitate both input and output. An empty editor ($\boxtimes$) stands for a prompt to input data, while a filled editor ($\square$) can be seen either as outputting a value, or as an input that comes with a default value.

Example 4.2.2 (Vending machine)

The following example demonstrates the use of external communication and choice.

We have a vending machine that dispenses a biscuit for one coin and a granola bar for two coins.

$\boxtimes \text{INT} \triangleright \lambda n. \text{if } n \equiv 1 \text{ then } \square \text{Biscuit} \text{ else if } n \equiv 2 \text{ then } \square \text{GranolaBar} \text{ else } \downarrow$

The editor $\boxtimes \text{INT}$ asks the user to enter an amount of money. This editor stands for a coin slot in a real machine that freely accepts and returns coins. There is a continue button on the machine, which sends a continue event to the step combinator ($\triangleright$). The button is initially disabled, due to the fact that the editor has no value. When the user has inserted exactly 1 or 2 coins, the continue button becomes enabled. When the user presses the continue button, the machine dispenses either a biscuit or a chocolate bar, depending on the amount of money. Snacks are modelled using a custom type.

4.2.4 Tasks can be observed

Several observations can be made on tasks. One of those is determining the value of a task. Not all tasks have a value, which makes value observation partial. I.e., the value of $\square 7$ is 7, but the value of $\boxtimes \text{INT}$ is $\perp$.

Another observation is the set of input events a task can respond to. For example, the task $\square 7$ can respond to value events, as discussed before.

To render a task, we need to observe a task's user interface. This is done compositionally. User interfaces of combined tasks are composed of the user interfaces of the components. For example, if two tasks combined with a step combinator, only the left-hand side is rendered. Two parallel tasks are rendered next to each other. Combining this information with the task's value and possible inputs, we can display the current state of the task, together with buttons that show the actions a user can engage in.

The final observation is to determine whether a task results in a failure, denoted by $\downarrow$. The step combinator $\triangleright$ and the choice combinator $\diamond$ use this to prevent users from picking a failing task.

4.2.5 Tasks are never done

Tasks never terminate, they always keep reacting to events. Editors can always be changed, and step combinators move on to new tasks.

In a step $t \triangleright e$, the decision to move on from a task t to its continuation e is taken by $\triangleright$, not by t . The decision is based on a speculative evaluation of e . The step combinator in $t \triangleright e$ passes the value v of t to the continuation e . Steps act like t as long as the step is guarded. A step is guarded if either the

left task has no value, or the speculative evaluation of e applied to v yields the failure task $\perp$. Once it becomes unguarded, the step continues as the result of $e\ v$. Speculative evaluation is designed so that possible side effects are undone.

Step combinators give rise to a form of internal communication. They represent data flow that *follows* control flow.

4.2.6 Tasks can share information

The step combinator is one form of internal communication, where task values are passed to continuations. Another form of internal communication is shared data. Shared data enables data flow *across* control flow, in particular between parallel tasks. Shared data sources are assignable references whose changes are immediately visible to all tasks interested in them. Users cannot directly interact with shared data, a shared editor is required for that. If x is a reference of type τ , then $\blacksquare x$ is an editor whose value is that of x .

The semantics of TOP requires all updates to shared data and all enabled internal steps to be processed before any further communication with the environment can take place.

Example 4.2.3 (Cigarette smokers)

The cigarette smokers problem by Downey (2008) is a surprisingly tricky synchronisation problem. We study it here because it demonstrates the capabilities of guarded steps. The problem is stated as follows. To smoke a cigarette, three ingredients are required: tobacco, paper, and a match. There are three smokers, each having one of the ingredients and requiring the other two. There is an agent that randomly provides two of those. The difficulty lies in the requirement that only the smoker may proceed whose missing ingredients are present.

Downey models availability of the ingredients with a semaphore for each ingredient. The agent randomly signals two of the three. The solution proposed by Downey involves an additional mutex, three additional semaphores, three additional threads called pushers, and three regular Boolean variables. The job of the pushers is to record availability of their ingredient in their Boolean variable, and check availability of other resources, waking the correct smoker when appropriate.

*The solution to this problem, essentially deadlock-free waiting for two semaphores, requires a substantial amount of additional synchronisation, together with non-trivial conditional statements. TOP allows a simple solution to this problem, using guarded steps. Steps can be guarded with arbitrary expressions. The parallel combinator can be used to watch two shared editors at the same time. Let *match*, *paper*, and *tobacco* be references to Booleans. The smokers are defined as follows.*

When the agent supplies two of the ingredients by setting the respective shares to True, only the step of the smoker that waits for those becomes enabled.

```

let continue =  $\lambda \langle x, y \rangle . \text{if } x \wedge y \text{ then smoke else } \zeta$  in
let tobaccoSmoker = ( $\blacksquare$  match  $\bowtie$   $\blacksquare$ paper)  $\triangleright$  continue in
let paperSmoker = ( $\blacksquare$  tobacco  $\bowtie$   $\blacksquare$ match)  $\triangleright$  continue in
let matchSmoker = ( $\blacksquare$  tobacco  $\bowtie$   $\blacksquare$ paper)  $\triangleright$  continue in
tobaccoSmoker  $\bowtie$  paperSmoker  $\bowtie$  matchSmoker

```

4.2.7 Tasks are predictable

Let t_1 and t_2 be tasks. The parallel combination $t_1 \bowtie t_2$ stands for two independent tasks carried out at the same time. This operator introduces interleaving concurrency. For the system it does not matter if the tasks are executed by two different people, or by one person who switches between the tasks. The inputs sent to the component tasks are interleaved into a serial stream, which is sent to the parallel combinator. We assume that such a serialisation is always possible. The tasks are truly independent of each other, if all interleavings are possible. The environment prefixes events to t_1 and t_2 respectively by F (first) and S (second). This unambiguously renames the inputs, removing any source of nondeterminism.

With concurrency comes the need for synchronisation, in situations where only some but not all interleavings are possible. The basic method for synchronisation in TOP is built into the step combinator. The task $t \triangleright e$ can only continue execution when two conditions are met: Task t must have a value v , and $e \ v$ must not evaluate to ζ . Programmers can encode arbitrary conditions in $e \ v$, which are evaluated atomically between interaction steps. This allows a variety of synchronisation problems to be solved in an intuitive and straight-forward manner.

C. A. R. Hoare (1985) states that nondeterminism is only ever useful for *specifying* systems, never for implementing them. TOP is meant solely for implementation and does not have any form of nondeterminism.

4.3 Conclusion

Collaboration in the real world consists of three aspects: communication, concurrency, and synchronisation. These aspects are reflected in TOP on a high level of abstraction, hiding the details of communication. For example, the cigarette smokers communicate with each other, but the programs do not explicitly mention sending or receiving events.

By focusing on collaboration instead of communication, TOP leads to directly executable specifications closer to real-world workflows which, at

the same time, can be used to generate multi-user applications to support such workflows.

All abilities described in the previous section are captured by the three building blocks of tasks: *editors*, *composition*, and *shared data*. The editors facilitate interaction and are the observable part of tasks. The combinators are at the heart of modelling collaboration. They describe what needs to be done, in which order. Finally, the shared data system facilitates communication between tasks. These three building blocks will be formalised in the next chapter.

Chapter 5

TopHat

*TOP implementations have been around for more than a decade, in the forms of *iTasks* and *mTasks*. These languages were developed with practical applicability in mind, and their semantics have only been given in the form of a reference implementation. TOP has been applied in projects with the Dutch coast guard, tax office, and navy. In those domains, it is vital to work with reliable software. The preferred way to verify that software works as expected, is by using formal methods.*

*Formal methods are mathematical techniques for verifying that applications are correct. However, to apply these techniques, we require a formal semantics for TOP. Currently, no TOP implementation has been formalised. This chapter decomposes the rich TOP features into elementary language elements, which makes them suitable for formal treatment. The simply typed lambda-calculus, extended with pairs, lists and references, is used as a base language. On top of this language, TopHat ($\widehat{\text{TOP}}$) is formalised, a language for modular interactive workflows. TopHat is described by means of a layered semantics. These layers consist of multiple big-step evaluations of expressions, and two labelled transition systems that handle user inputs. With $\widehat{\text{TOP}}$, the foundation is laid for formal reasoning over TOP languages and programs. On top of that, having a formal semantics allows us to better compare $\widehat{\text{TOP}}$ with other languages that model and coordinate collaboration. $\widehat{\text{TOP}}$ has been implemented in Haskell, and the task layer on top of the *iTasks* framework. By developing a formal semantics, and implementing the semantics, we demonstrate that it is indeed feasible to develop a usable, formal TOP language. Having a TOP language with a formal semantics matters because formal program verification is important for mission-critical software, especially for systems with concurrency.*

5.1 Introduction

This chapter presents TopHat ($\widehat{\text{TOP}}$), a Task-oriented Programming (TOP) language with a formal semantics. $\widehat{\text{TOP}}$ is based on the TOP constructs presented in Chapter 4. It consists of a task language that is embedded

in a simply typed lambda calculus. First, the language constructs are presented, followed by some illustrative examples. Then the formal semantics is presented. To verify our approach, we prove several properties for these semantics, and we present a practical implementation. With this chapter, we lay the ground work for the application of formal methods to reason about TOP software. In Chapter 6, we will use the formal semantics of $\widehat{\text{TOP}}$ to develop a symbolic execution semantics.

5.2 Language

In this section, we present the constructs of $\widehat{\text{TOP}}$, our modular interactive workflow language. We define the host and task language, the types, and the static semantics. Then we describe the workings of each construct using examples. These constructs are formalised in Section 5.4.

5.2.1 Expressions

The host language is a simply typed λ -calculus, extended with some basic types and ML-style references. We use references to represent shared data sources. The simply typed λ -calculus does not support recursion. The grammar in Fig. 5.1 defines the syntax of the host language. It has abstractions, applications, variables, and constants for booleans, integers and strings. The symbol $\star$ stands for binary operators. For the result of parallel tasks we need pairs. Conditionals come in handy for defining guards. References will be used to implement shared editors. Our treatment of references closely follows the one by Pierce (2002). Creating a reference using the keyword **ref** yields a location l . x denotes program variables, l denotes store locations. Locations are not intended to be directly manipulated by the programmer. The symbols $!$ and $:=$ stand for dereferencing and assignment. The unit value is used as the result of assignments.

Notation We use double quotation marks to denote strings. Integers are denoted by their numerical representation, and booleans are written `True` and `False`. We freely make use of the logic operators $\neg$, $\wedge$, and $\vee$, arithmetic operators $+$, $-$, $\times$, $/$, and the string append operator $++$. Furthermore, we use standard comparison operations $<$, $\leq$, $\equiv$, $\neq$, $\geq$, and $>$. The symbol $\star$ stands for any of those. The notation $e_1; e_2$ is an abbreviation for $(\lambda x : \text{UNIT}. e_2) e_1$, where x is a fresh variable. The notation **let** $x : \tau = e_1$ **in** e_2 is an abbreviation for $(\lambda x : \tau. e_2) e_1$.

Expressions

e	$::=$	$\lambda x : \tau. e \mid e_1 e_2 \mid x$	– abstraction, application, variable
		$\mid c \mid \langle \rangle \mid u e_1 \mid e_1 o e_2$	– constant, unit, unary, binary operation
		$\mid \mathbf{if} e_1 \mathbf{then} e_2 \mathbf{else} e_3$	– conditional
		$\mid \langle e_1, e_2 \rangle \mid \mathbf{fst} e \mid \mathbf{snd} e$	– pair, projections
		$\mid []_\beta \mid e_1 :: e_2$	– nil, cons
		$\mid \mathbf{head} e \mid \mathbf{tail} e$	– first element, list tail
		$\mid \mathbf{ref} e \mid !e \mid e_1 := e_2 \mid l$	– references, location
		$\mid p$	– pretask

Constants

c	$::=$	$B \mid I \mid S$	– boolean, integer, string
-----	-------	-------------------	----------------------------

Unary Operations

u	$::=$	$\neg \mid - \mid \mathbf{len} \mid \mathbf{uniq}$	– not, negate, length, unique
-----	-------	--	-------------------------------

Binary Operations

o	$::=$	$< \mid \leq \mid \equiv \mid \neq \mid \geq \mid >$	– equational
		$\mid + \mid - \mid \times \mid /$	– numerical
		$\mid \wedge \mid \vee$	– boolean
		$\mid ++ \mid \in$	– append, element of

FIGURE 5.1: Language grammar

Pretasks		
$p ::=$	$\square e \mid \boxtimes \beta \mid \blacksquare e$	– valued editor, unvalued editor, shared editor
	$\mid e_1 \blacktriangleright e_2 \mid e_1 \triangleright e_2$	– internal step, external step
	$\mid \text{fail} \mid e_1 \bowtie e_2$	– fail, parallel composition
	$\mid e_1 \blacklozenge e_2 \mid e_1 \lozenge e_2$	– internal choice, external choice

FIGURE 5.2: Task grammar

Types		
$\tau ::=$	$\tau_1 \rightarrow \tau_2 \mid \beta \mid \text{REF } \tau \mid \text{TASK } \tau$	– function, basic, reference, task
Basic types		
$\beta ::=$	$\tau_1 \times \tau_2 \mid \text{LIST } \beta \mid \text{UNIT}$	– product, list, unit
	$\mid \text{BOOL} \mid \text{INT} \mid \text{STRING}$	– boolean, integer, string

FIGURE 5.3: Type grammar

Pretasks The grammar in Fig. 5.2 specifies the syntactic category of *pre-tasks*. Pretasks are tasks that contain unevaluated subexpressions. Each pretask will be discussed in more detail in the following subsections. We use open symbols ($\square, \boxtimes, \triangleright, \lozenge$) for tasks that require user input, and closed symbols ($\blacksquare, \blacktriangleright, \blacklozenge$) for tasks that can be evaluated without user input.

Typing Fig. 5.3 shows the grammar of types used by $\widehat{\text{TOP}}$. It has functions, pairs, basic types, unit, references, and tasks.

The typing rules for expressions are given in Fig. 5.4. Most typing rules lift the type of their subexpressions into the Task-type. The typing rules for steps make sure the continuations e_2 are functions that accept a well-typed value from the left-hand side (T-THEN, T-NEXT). References, and therefore shared editors, can only be of a basic type so they do not introduce implicit recursion (T-UPDATE).

5.2.2 Editors

Programs in $\widehat{\text{TOP}}$ model interactive workflows. Interaction means communication with end users. End users should be able to enter information into the system, change it, clear it, reenter it, and so on. To do this, we introduce the concept of *editors*. Editors are typed containers that either hold a value or are empty. Editors that have a value can be *changed*. Empty editors can be *filled*.

$\Gamma, \Sigma \vdash e : \tau$			
T-CONSTBOOL $c \in B$ $\Gamma, \Sigma \vdash c : \text{Bool}$	T-CONSTINT $c \in I$ $\Gamma, \Sigma \vdash c : \text{Int}$	T-CONSTSTRING $c \in S$ $\Gamma, \Sigma \vdash c : \text{String}$	T-VAR $x : \tau \in \Gamma$ $\Gamma, \Sigma \vdash x : \tau$
T-UNIT $\Gamma, \Sigma \vdash \langle \rangle : \text{Unit}$	T-LOC $\Sigma(l) = \beta$ $\Gamma, \Sigma \vdash l : \text{Ref } \beta$	T-PAIR $\Gamma, \Sigma \vdash e_1 : \tau_1 \quad \Gamma, \Sigma \vdash e_2 : \tau_2$ $\Gamma, \Sigma \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2$	
T-FIRST $\Gamma, \Sigma \vdash e : \tau_1 \times \tau_2$ $\Gamma, \Sigma \vdash \text{fst } e : \tau_1$	T-SECOND $\Gamma, \Sigma \vdash e : \tau_1 \times \tau_2$ $\Gamma, \Sigma \vdash \text{snd } e : \tau_2$	T-LISTEMPTY $\Gamma, \Sigma \vdash []_\beta : \text{List } \beta$	
T-LISTCONS $\Gamma, \Sigma \vdash e_1 : \beta$ $\Gamma, \Sigma \vdash e_2 : \text{List } \beta$ $\Gamma, \Sigma \vdash e_1 :: e_2 : \text{List } \beta$	T-LISTHEAD $\Gamma, \Sigma \vdash e : \text{List } \beta$ $\Gamma, \Sigma \vdash \text{head } e : \beta$	T-LISTTAIL $\Gamma, \Sigma \vdash e : \text{List } \beta$ $\Gamma, \Sigma \vdash \text{tail } e : \text{List } \beta$	
T-IF T-ABS $\Gamma[x : \tau_1], \Sigma \vdash e : \tau_2$ $\Gamma, \Sigma \vdash \lambda x : \tau_1. e : \tau_1 \rightarrow \tau_2$ $\Gamma, \Sigma \vdash e_1 : \text{Bool}$ $\Gamma, \Sigma \vdash e_2 : \tau$ $\Gamma, \Sigma \vdash e_3 : \tau$ $\Gamma, \Sigma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau$			
T-APP $\Gamma, \Sigma \vdash e_1 : \tau_1 \rightarrow \tau_2$ $\Gamma, \Sigma \vdash e_2 : \tau_1$ $\Gamma, \Sigma \vdash e_1 e_2 : \tau_2$	T-REF $\Gamma, \Sigma \vdash e : \beta$ $\Gamma, \Sigma \vdash \text{ref } e : \text{Ref } \beta$	T-DEREF $\Gamma, \Sigma \vdash e : \text{Ref } \beta$ $\Gamma, \Sigma \vdash !e : \beta$	
T-ASSIGN $\Gamma, \Sigma \vdash e_1 : \text{Ref } \beta$ $\Gamma, \Sigma \vdash e_2 : \beta$ $\Gamma, \Sigma \vdash e_1 := e_2 : \text{Unit}$	T-FAIL $\Gamma, \Sigma \vdash \text{!} : \text{Task } \tau$	T-EDIT $\Gamma, \Sigma \vdash e : \beta$ $\Gamma, \Sigma \vdash \square e : \text{Task } \beta$	
T-ENTER $\Gamma, \Sigma \vdash \boxtimes \beta : \text{Task } \beta$	T-UPDATE $\Gamma, \Sigma \vdash e : \text{Ref } \beta$ $\Gamma, \Sigma \vdash \blacksquare e : \text{Task } \beta$	T-OR $\Gamma, \Sigma \vdash e_1 : \text{Task } \tau$ $\Gamma, \Sigma \vdash e_2 : \text{Task } \tau$ $\Gamma, \Sigma \vdash e_1 \blacklozenge e_2 : \text{Task } \tau$	
T-THEN $\Gamma, \Sigma \vdash e_1 : \text{Task } \tau_1$ $\Gamma, \Sigma \vdash e_2 : \tau_1 \rightarrow \text{Task } \tau_2$ $\Gamma, \Sigma \vdash e_1 \blacktriangleright e_2 : \text{Task } \tau_2$	T-NEXT $\Gamma, \Sigma \vdash e_1 : \text{Task } \tau_1$ $\Gamma, \Sigma \vdash e_2 : \tau_1 \rightarrow \text{Task } \tau_2$ $\Gamma, \Sigma \vdash e_1 \triangleright e_2 : \text{Task } \tau_2$		
T-AND $\Gamma, \Sigma \vdash e_1 : \text{Task } \tau_1$ $\Gamma, \Sigma \vdash e_2 : \text{Task } \tau_2$ $\Gamma, \Sigma \vdash e_1 \bowtie e_2 : \text{Task } (\tau_1 \times \tau_2)$	T-XOR $\Gamma, \Sigma \vdash e_1 : \text{Task } \tau$ $\Gamma, \Sigma \vdash e_2 : \text{Task } \tau$ $\Gamma, \Sigma \vdash e_1 \diamond e_2 : \text{Task } \tau$		

FIGURE 5.4: Typing rules

Editors are used for various forms of input and output, for example widgets in a GUI, form fields on a webpage, sensors, or network connections. Consider an editor for a person's age on a web page. Users can change the value until they are satisfied with it. Editors are meant to capture this constantly changing nature of user input. The user interface of an editor depends on its type. This could be an input field for strings, a toggle switch for booleans, or even a map with a pin for locations.

Valued and unvalued editors ($\square e, \boxtimes \beta$) Editors that hold an expression $e : \beta$ have type $\text{Task } \beta$. Empty editors are annotated with a type to ensure type safety and type preservation during evaluation.

Shared editors ($\blacksquare e$) Shared editors watch references, lifting their value into the task domain. If e is a reference $\text{REF } \beta$, then $\blacksquare e$ is of type $\text{Task } \beta$.

Changes to a shared editor are immediately visible to all shared editors watching the same reference. Imagine two users, Marco and Christopher, both watching shared editors of the same coordinates. The editors are visualised as a pin on a map. When Marco moves his pin, he updates the value of the shared editor, thereby changing the value of the reference. This change is immediately reflected on Christopher's screen: The pin changes its position on his map. This way Marco and Christopher can work together to edit the same information.

Two other important use cases for shared editors are sensors and time. Sensors can be represented as external entities that periodically update a shared editor with their current sensor value. Similarly, the current time can be stored in a shared editor ($\blacksquare \text{time}$) that is periodically updated by a clock. The actual sensor and the clock are not modelled in $\widehat{\text{TOP}}$. We assume that they exist as external users that send update events to the system. This allows programmers to write tasks that react to sensor values or timeouts.

5.2.3 Steps

Editors represent atomic units of work. In this section we look at ways to compose smaller tasks into bigger ones. Composing tasks can be done in two ways, sequential and parallel. Parallel composition comes in two variants: combining two tasks (*and-parallel*) and choosing between two tasks (*or-parallel*). We study sequential composition first, and after that combining and choosing.

Internal and external step ($t \blacktriangleright e, t \triangleright e$) Sequential composition has a task t on the left and a continuation e on the right. External steps ($\triangleright$) are triggered

by the user, while internal steps ($\blacktriangleright$) are taken automatically. The accompanying typing rules are T-THEN and T-NEXT. According to these rules, the left-hand side is a task $t : \text{TASK } \tau_1$, and the right hand side $e : \tau_1 \rightarrow \text{TASK } \tau_2$ is a function that, given the task value of t , calculates the task with which to continue.

Steps are guarded, which means that the step combinators can only proceed when the following conditions are met. The left-hand side must have a value, only then can the right hand side calculate the successor task. The successor task must not be ζ , introduced below. This is enforced on the semantic level, as described in the next section. The internal step can proceed immediately when these conditions are met. The external step must additionally receive a continue event C .

Example 5.2.1 (Conditional stepping)

Consider the following:

$\boxtimes \text{INT} \blacktriangleright \lambda n. \text{if } n \equiv 42 \text{ then } \square \text{"Good"} \text{ else } \square \text{"Bad"}$

Initially, the step is guarded because the editor does not have a value. When a user enters an integer, the program continues immediately with either $\square \text{"Good"}$ or $\square \text{"Bad"}$, depending on the input.

Fail (ζ) Fail is a task that never has a value and never accepts input. The typing rule T-FAIL states that it has type $\text{TASK } \tau$ for any type τ . Programmers can use ζ to tell steps that no sensible successor task can proceed.

Example 5.2.2 (Guarded stepping)

Consider this slight variation on Example 5.2.1:

$\boxtimes \text{INT} \blacktriangleright \lambda n. \text{if } n \equiv 42 \text{ then } \square \text{"Good"} \text{ else } \zeta$

The user is asked to enter an integer. As long as the right hand side of $\blacktriangleright$ evaluates to ζ , the step cannot proceed, and the user can keep editing the integer. As soon as the value of the left-hand side is 42, the right hand side evaluates to something other than ζ , and the step proceeds to $\square \text{"Good"}$.

Example 5.2.3 (Waiting)

With the language constructs seen so far it is possible to create a task that waits for a specified amount of time. To do this, we make use of a shared editor holding the current time (see Section 5.2.2), and a guarded internal step.

let $wait : INT \rightarrow TASK\ UNIT = \lambda amount : INT.$

time $\blacktriangleright \lambda start : INT.$

time $\blacktriangleright \lambda now : INT.$

if $now > start + amount$ **then** $\square \langle \rangle$ **else** $\downarrow$

The first step is immediately taken, resulting in $start$ to be the time at the moment $wait$ is executed. The second step is guarded until the current time is greater to the start time plus the required amount.

5.2.4 Parallel

A common pattern in workflow design is splitting up work into multiple tasks that can be executed simultaneously. In $\widehat{TOP}$, all parallel branches can progress independently, driven by input events. This requires inputs to be tagged in order to reach the intended task.

There are two ways to proceed after a parallel composition. One way is to wait for all tasks to produce results and combine those, the other to pick the first available result. Both ways introduce explicit forks and implicit joins in $\widehat{TOP}$.

Combination ($e_1 \bowtie e_2$) A combination of two tasks is a parallel *and*. It has a value only if both branches have a value. This is reflected in the typing rule T-AND, It shows that if the first task has type τ_1 , and the second has type τ_2 , their combination has the pair type $\tau_1 \times \tau_2$.

Example 5.2.4 (Combining)

The task

$\boxtimes INT \bowtie \square "Batman" \blacktriangleright \lambda \langle n, s \rangle . \square (\text{replicate } n "Na" ++ s)$

can only step when both editors have values. When it steps, the continuation uses the pair to calculate the result.

Internal and external choice ($e_1 \blacklozenge e_2, e_1 \diamond e_2$) Internal choice ($\blacklozenge$) is a parallel *or*. It picks the leftmost branch that has a value. Its typing rule T-OR states that both branches must have the same type $TASK\ \tau$. For example, $\boxtimes INT \blacklozenge \square 37$ normalises to $\square 37$, because $\boxtimes INT$ doesn't have a value. Users can work on both branches of an internal choice simultaneously.

External choice ($\diamond$) is different in this regard. An external choice requires users to pick a branch before continuing with it. This means users cannot work on the branches of an external choice before picking one.

Example 5.2.5 (Delay)

We illustrate the use of internal and external choice by means of an example that asks a user to proceed with a given task or to cancel. If the user does not make a choice within a given time frame, the program proceeds automatically. The example makes use of the task *wait* from Example 5.2.3.

$$\begin{aligned} \text{let } \text{cancel} : \text{TASK UNIT} &= \square \langle \rangle \text{ in} \\ \text{let } \text{delay} : \text{INT} \rightarrow \text{TASK UNIT} \rightarrow \text{TASK UNIT} &= \lambda n. \lambda \text{proceed}. \\ &(\text{proceed} \diamond \text{cancel}) \blacklozenge (\text{wait } n \blacktriangleright \lambda u : \text{UNIT}. \text{proceed}) \end{aligned}$$

Note that *delay* is higher-order. It is a task that takes another task as parameter.

5.2.5 Annotations

Tasks can be annotated with additional information. The system can use this information in various ways. Possible use cases are labels for the user interface, resource consumption information for static resource analysis, or messages for automatic end-user feedback. Annotations are not covered in this chapter. Our Haskell implementation of $\widehat{\text{TOP}}$ supports annotating tasks with user IDs, so that individual tasks in a large workflow can be assigned to different users. These annotations are used to filter the user interfaces for each user so that they can only see their part of the workflow.

5.3 Example

In this section we develop an example program to demonstrate the capabilities of $\widehat{\text{TOP}}$. The example is a small flight booking system. It demonstrates communication on all three levels: with the environment, across control flow, and alongside it. Also, it shows synchronisation and input validation.

The requirements of the application are as follows. 1. A user has to enter a list of passengers for which to book tickets. 2. At least one of these passengers has to be an adult. 3. After a valid list of passengers has been entered, the user has to pick seats. 4. Only free seats may be picked. 5. Every passenger must have exactly one seat. 6. Multiple users should be able to book tickets at the same time.

For this example we assume that the host language has four functions over lists: *all*, *any*, *intersect*, and *difference*. The functions *all* and *any* check if all or any elements in a list satisfy a given predicate. The functions *intersect* and *difference* compute the set-intersection and set-difference of two lists.

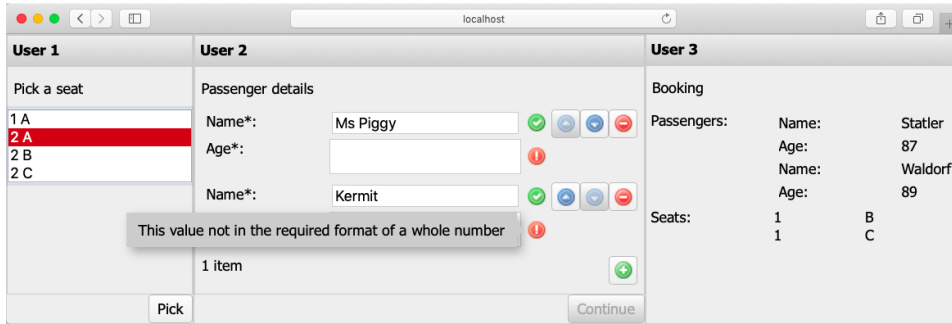


FIGURE 5.5: Running web application of the flight booking example using a translation to iTasks.

We also make use of string equality ($\equiv$), dereferencing (!), reference assignment ($:=$), and expression sequencing (;). For brevity, we omit the type annotations of variable bindings.

Example 5.3.1 (Flight booking)

We start off by defining some type aliases. A passenger is a pair with name and age. A seat is a pair with a row number and a seat letter.

type PASSENGER = STRING $\times$ INT

type SEAT = INT $\times$ STRING

Choosing seats requires reading and updating shared information. The list of free seats is stored in a reference.

let freeSeats = **ref** [$\langle 1, "A" \rangle$, $\langle 1, "B" \rangle$, $\langle 1, "C" \rangle$, ...]

Now we develop our workflow in a top-down manner. Our flight booking starts with an interactive task $\boxtimes$ (LIST PASSENGER), where users can enter a list of passengers. A task $\boxtimes \tau$ is an empty editor that asks for a value of the given type τ . Passengers are valid if their name is not empty and their age is at least 0. Lists of passengers are valid if each passenger is valid, and at least one of the passengers is an adult. When the user has entered a valid list of passengers, the step after $\triangleright$ becomes enabled, and the user can proceed to picking seats. In case of an invalid list of passengers, the step is guarded by the failing task $\downarrow$.

let valid = $\lambda p. \neg (fst\ p \equiv "") \wedge snd\ p \geq 0$ **in**

let adult = $\lambda p. snd\ p \geq 18$ **in**

let allValid = $\lambda ps. all\ valid\ ps \wedge any\ adult\ ps$ **in**

let bookFlight = $\boxtimes$ (LIST PASSENGER) $\triangleright \lambda ps.$

if allValid ps **then** chooseSeats ps **else** $\downarrow$

A selection of seats is correct if every entered seat is free.

```

let correct =  $\lambda ss.$  intersect ss !freeSeats  $\equiv$  ss in
let chooseSeats =  $\lambda ps.$   $\boxtimes$ (LIST SEAT)  $\triangleright$   $\lambda ss.$ 
  if correct ss  $\wedge$  length ps  $\equiv$  length ss
  then confirmBooking ps ss else  $\downarrow$ 

```

The function *confirmBooking* removes the selected seats from the shared list of free seats, and displays the end result using an editor, denoted by $\square$.

```

let confirmBooking =  $\lambda ps.$   $\lambda ss.$ 
  freeSeats := difference !freeSeats ss;  $\square\langle ps, ss \rangle$ 

```

The main task starts three *bookFlight* tasks, which could be performed by three different users in parallel.

```

bookFlight  $\bowtie$  bookFlight  $\bowtie$  bookFlight

```

A screenshot of the running application is shown in Fig. 5.5.

All instances of the *bookFlight* task have access to the shared list of free seats. Rewriting the example in a language without side effects would not only be cumbersome, obfuscating the code with explicit threading of state, but it would be impossible to model the parallel execution of three *bookFlight* tasks. It is not known upfront which task will finish first, and thus it is not possible to thread the free seat list between the parallel tasks.

5.4 Semantics

In this section we formalise the semantics of the language constructs described in Section 5.2. We organise this by following the structure of the language. Firstly, the task language is embedded in a simply typed λ -calculus. This requires a specification of the *evaluation* of terms in the host language, and how it handles the task language. Secondly, there are two ways to drive evaluation of task expressions, internally by the system itself, and externally by the user. This is done in two additional semantics, one for the internal *normalisation* of tasks, and another for the *interaction* with the end user.

The three main layers of semantics are thus evaluation, normalisation, and interaction. The semantics, together with *observations*, will be discussed in the following subsections. Fig. 5.6 shows the relation between all semantics arrows. It also shows that there are two helper semantics, *handle* and

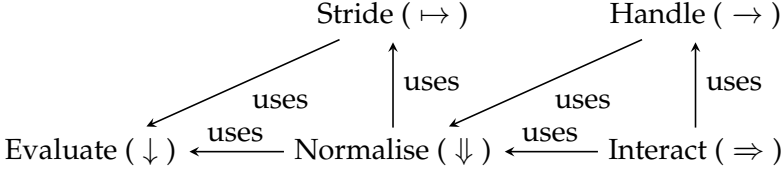


FIGURE 5.6: Semantic functions defined in this report and their relation.

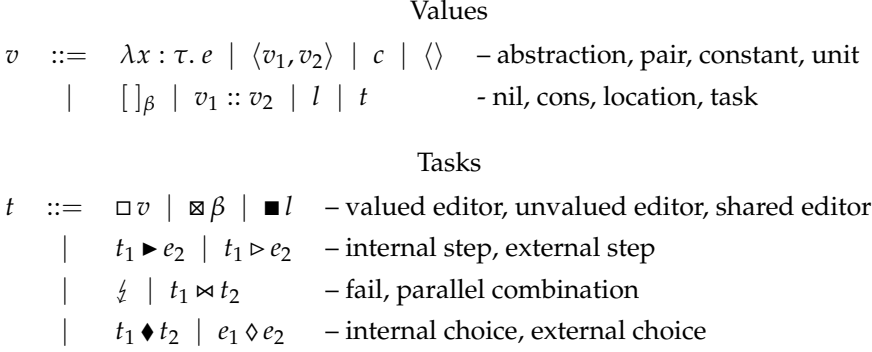


FIGURE 5.7: Value grammar

stride. We use the convention that downward arrows are big-step semantics, and rightward arrows are small-step semantics. One of our explicit goals is to keep the semantics for evaluation and normalisation separate, to not mix general purpose programming notions with workflow specific semantics. This is achieved by letting tasks be values in the host language.

5.4.1 Evaluating expressions

The host language evaluates expressions using a big-step semantics. To ease reasoning about references, we choose a call-by-value evaluation strategy.

Fig. 5.7 shows values that are the result of the evaluation semantics. Tasks are values, and the operands of task constructors are evaluated eagerly. Exceptions to this are steps and external choice, where some or all of the operands are not evaluated.

The rules to evaluate expressions e are listed in Fig. 5.8. Most rules do not differ from standard rules for evaluating expressions in the simply typed lambda calculus (Pierce, 2002), except for the task constructs. The evaluation rules for tasks can be deduced from the value grammar. Most task constructors are strict in their arguments. Only steps keep their right hand side unevaluated to delay side effects till the moment the step is taken. The same holds for both branches of the external choice.

$e, \sigma \downarrow v, \sigma'$		
E-VALUE $\frac{}{v, \sigma \downarrow v, \sigma'}$	E-PAIR $\frac{e_1, \sigma \downarrow v_1, \sigma' \quad e_2, \sigma' \downarrow v_2, \sigma''}{\langle e_1, e_2 \rangle, \sigma \downarrow \langle v_1, v_2 \rangle, \sigma''}$	
E-FIRST $\frac{e, \sigma \downarrow \langle v_1, v_2 \rangle, \sigma'}{\text{fst } e, \sigma \downarrow v_1, \sigma'}$	E-SECOND $\frac{e, \sigma \downarrow \langle v_1, v_2 \rangle, \sigma'}{\text{snd } e, \sigma \downarrow v_2, \sigma'}$	
E-HEAD $\frac{e, \sigma \downarrow v_1 :: v_2, \sigma'}{\text{head } e, \sigma \downarrow v_1, \sigma'}$	E-CONS $\frac{e_1, \sigma \downarrow v_1, \sigma' \quad e_2, \sigma' \downarrow v_2, \sigma''}{e_1 :: e_2, \sigma \downarrow v_1 :: v_2, \sigma''}$	
E-TAIL $\frac{e, \sigma \downarrow v_1 :: v_2, \sigma'}{\text{tail } e, \sigma \downarrow v_2, \sigma'}$	E-APP $\frac{e_1, \sigma \downarrow \lambda x : \tau. e'_1, \sigma' \quad e_2, \sigma' \downarrow v_2, \sigma''}{e'_1[x \mapsto v_2], \sigma'' \downarrow v_1, \sigma'''}$ $\frac{}{e_1 e_2, \sigma \downarrow v_1, \sigma'''}$	
E-IFTRUE $\frac{e_1, \sigma \downarrow \text{True}, \sigma' \quad e_2, \sigma' \downarrow v_2, \sigma''}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3, \sigma \downarrow v_2, \sigma''}$	E-IFFALSE $\frac{e_1, \sigma \downarrow \text{False}, \sigma' \quad e_3, \sigma' \downarrow v_3, \sigma''}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3, \sigma \downarrow v_3, \sigma''}$	
E-REF $\frac{e, \sigma \downarrow v, \sigma' \quad l \notin \text{Dom}(\sigma')}{\text{ref } e, \sigma \downarrow l, \sigma'[l \mapsto v]}$	E-DEREF $\frac{e, \sigma \downarrow l, \sigma'}{!e, \sigma \downarrow \sigma'(l), \sigma'}$	
E-ASSIGN $\frac{e_1, \sigma \downarrow l, \sigma' \quad e_2, \sigma' \downarrow v_2, \sigma''}{e_1 := e_2, \sigma \downarrow \langle \rangle, \sigma''[l \mapsto v_2]}$	E-EDIT $\frac{e, \sigma \downarrow v, \sigma'}{\square e, \sigma \downarrow \square v, \sigma'}$	
E-UPDATE $\frac{e, \sigma \downarrow l, \sigma'}{\blacksquare e, \sigma \downarrow \blacksquare l, \sigma'}$	E-THEN $\frac{e_1, \sigma \downarrow t_1, \sigma'}{e_1 \blacktriangleright e_2, \sigma \downarrow t_1 \blacktriangleright e_2, \sigma'}$	E-NEXT $\frac{e_1, \sigma \downarrow t_1, \sigma'}{e_1 \triangleright e_2, \sigma \downarrow t_1 \triangleright e_2, \sigma'}$
E-AND $\frac{e_1, \sigma \downarrow t_1, \sigma' \quad e_2, \sigma' \downarrow t_2, \sigma''}{e_1 \bowtie e_2, \sigma \downarrow t_1 \bowtie t_2, \sigma''}$	E-OR $\frac{e_1, \sigma \downarrow t_1, \sigma' \quad e_2, \sigma' \downarrow t_2, \sigma''}{e_1 \blacklozenge e_2, \sigma \downarrow t_1 \blacklozenge t_2, \sigma''}$	

FIGURE 5.8: Evaluation semantics for expressions

$$\begin{aligned}
\mathcal{V} &: \text{Task} \times \text{State} \rightharpoonup \text{Value} \\
\mathcal{V}(\Box v, \sigma) &= v \\
\mathcal{V}(\boxtimes \beta, \sigma) &= \perp \\
\mathcal{V}(\blacksquare l, \sigma) &= \sigma(l) \\
\mathcal{V}(\downarrow, \sigma) &= \perp \\
\mathcal{V}(t_1 \blacktriangleright e_2, \sigma) &= \perp \\
\mathcal{V}(t_1 \triangleright e_2, \sigma) &= \perp \\
\mathcal{V}(t_1 \bowtie t_2, \sigma) &= \begin{cases} \langle v_1, v_2 \rangle & \text{when } \mathcal{V}(t_1, \sigma) = v_1 \wedge \mathcal{V}(t_2, \sigma) = v_2 \\ \perp & \text{otherwise} \end{cases} \\
\mathcal{V}(t_1 \blacklozenge t_2, \sigma) &= \begin{cases} v_1 & \text{when } \mathcal{V}(t_1, \sigma) = v_1 \\ v_2 & \text{when } \mathcal{V}(t_1, \sigma) = \perp \wedge \mathcal{V}(t_2, \sigma) = v_2 \\ \perp & \text{otherwise} \end{cases} \\
\mathcal{V}(t_1 \lozenge t_2, \sigma) &= \perp
\end{aligned}$$

FIGURE 5.9: Values observation function.

5.4.2 Task observations

The normalisation ($\Downarrow$) and interaction ($\Rightarrow$) semantics make use of observations on tasks. Observations are semantic functions on the syntax tree of tasks. There are four semantic functions: $\mathcal{V}$ for the current task value, $\mathcal{F}$ to determine if a task fails, $\mathcal{I}$ for the currently accepted input events, and a function for generating user interfaces. The semantics make use of $\mathcal{V}$ and $\mathcal{F}$, while $\mathcal{I}$ is used for proving safety. The function for user interfaces is not used by the semantics, but by our implementation. It is only described in passing here.

Observable values ($\mathcal{V}$) Task values are used by steps to calculate the successor task. Filled editors are tasks that contain values, as are shared editors. Unvalued editors do not contain values, neither does the fail task. These facts propagate through all other task constructors. The function $\mathcal{V}$ associates a value v to task t where possible. Its definition is given in Fig. 5.9. We use a half arrow ($\rightharpoonup$) to indicate that this function is partial, and $\perp$ to indicate when the function is undefined.

Internal and external steps do not have an observable value, because calculating the value would require evaluation of the continuation. Parallel composition only has a value when both branches have values, in which case these values are paired. Internal choice has a value when one of the branches has a value. When both branches have a value, it takes the value of the left branch. External choice does not have a value because it waits for user input.

$$\begin{aligned}
\mathcal{F} : \text{Task} \times \text{State} &\rightarrow \text{Bool} \\
\mathcal{F} (\Box v, \sigma) &= \text{False} \\
\mathcal{F} (\boxtimes \beta, \sigma) &= \text{False} \\
\mathcal{F} (\blacksquare l, \sigma) &= \text{False} \\
\mathcal{F} (\zeta, \sigma) &= \text{True} \\
\mathcal{F} (t_1 \blacktriangleright e_2, \sigma) &= \mathcal{F} (t_1, \sigma) \\
\mathcal{F} (t_1 \triangleright e_2, \sigma) &= \mathcal{F} (t_1, \sigma) \\
\mathcal{F} (t_1 \bowtie t_2, \sigma) &= \mathcal{F} (t_1, \sigma) \wedge \mathcal{F} (t_2, \sigma) \\
\mathcal{F} (t_1 \blacklozenge t_2, \sigma) &= \mathcal{F} (t_1, \sigma) \wedge \mathcal{F} (t_2, \sigma) \\
\mathcal{F} (e_1 \diamond e_2, \sigma) &= \mathcal{F} (t_1, \sigma'_1) \wedge \mathcal{F} (t_2, \sigma'_2) \\
&\quad \textbf{where } e_1, \sigma \Downarrow t_1, \sigma'_1 \textbf{ and } e_2, \sigma \Downarrow t_2, \sigma'_2
\end{aligned}$$

FIGURE 5.10: Failing observation function.

Failing ($\mathcal{F}$) ζ stands for an impossible task. Combinations of tasks can also be impossible. Take for example the parallel composition of two fails ($\zeta \bowtie \zeta$). This expression is equivalent to ζ , because it cannot handle input and cannot be further normalised. A task is considered impossible if there is no interaction possible, and there is no observable task value. This intuition is formalised by the function $\mathcal{F}$ in Fig. 5.10. It determines whether a task is impossible. Such tasks are called *failing*.

Steps whose left-hand sides are failing can never proceed because of the lack of an observable value. Therefore, they are itself failing. The parallel combination of two tasks is failing if they are both failing. If only one of them fails, there is still interaction possible with the other task. The internal choice of two failing tasks is failing. External choices let the user pick a side and only then evaluate the corresponding subexpression. To determine if an external choice is failing, it needs to peek into the future to check if both subexpressions are failing. The choice case relies on the normalisation semantics ($\Downarrow$) defined in the next section.

User interface $\widehat{\text{TOP}}$ is designed such that a user interface can be generated from a task's syntax tree. A possible graphical user interface is shown in Fig. 5.5, where tasks are rendered as HTML pages. Editors are rendered as input fields, external choices are represented by two buttons, and parallel tasks are rendered side by side. Steps only show the interface of their left-hand side. In case of an external step they are accompanied by a button. When the guard condition of a step is not fulfilled, the button is disabled.

$$\boxed{e, \sigma \Downarrow t, \sigma'}$$

N-DONE $ \frac{ \begin{array}{l} e, \sigma \Downarrow t, \sigma' \\ t, \sigma' \mapsto t', \sigma'' \\ \sigma' = \sigma'' \wedge t = t' \end{array} }{ e, \sigma \Downarrow t, \sigma' } $	N-REPEAT $ \frac{ \begin{array}{l} e, \sigma \Downarrow t, \sigma' \\ t, \sigma' \mapsto t', \sigma'' \\ \sigma' \neq \sigma'' \vee t \neq t' \\ t', \sigma'' \Downarrow t'', \sigma''' \end{array} }{ e, \sigma \Downarrow t'', \sigma''' } $
--	--

FIGURE 5.11: Normalisation semantics

5.4.3 Normalising tasks

The normalisation semantics is responsible for reducing expressions of type `TASK` until they are ready to handle input. It is a big-step semantics, and makes use of evaluation of the host language. We write $e, \sigma \Downarrow t, \sigma'$ to describe that an expression e in state σ normalises to task t in state σ' .

Normalisation rules are given in Fig. 5.11. Both rules ensure that expressions are first evaluated by the host language ($\Downarrow$), and then by the stride semantics ($\mapsto$). These two actions are repeated until the resulting state and task stabilise.

The striding semantics is responsible for reducing internal steps and internal choices. A stride from task t in state σ to t' in state σ' is denoted by $t, \sigma \mapsto t', \sigma'$. The rules for striding are given in Fig. 5.12. Tasks like editors, fail and external choice are not further reduced. For external choice and parallel there are congruence rules.

The split between striding and normalisation is due to mutable references. Consider the following example, where $\sigma = \{l \mapsto \text{False}\}$.

$$(\blacksquare l \blacktriangleright \lambda x:\text{Bool}. \text{if } x \text{ then } e \text{ else } \frac{1}{2}) \bowtie (l := \text{True}; \square \langle \rangle)$$

S-AND reduces this expression in one step to

$$(\blacksquare l \blacktriangleright \lambda x:\text{Bool}. \text{if } x \text{ then } e \text{ else } \frac{1}{2}) \bowtie (\square \langle \rangle)$$

with $\sigma' = \{l \mapsto \text{True}\}$. This expression is not normalised, because the left task can take a step. The issue here lies in the fact that the right task updates l . The N-DONE and N-REPEAT rules ensure that striding is applied until the state σ becomes stable and no further normalisation can take place.

	$t, \sigma \mapsto t', \sigma'$
S-EDIT	S-THENSTAY
$\square v, \sigma \mapsto \square v, \sigma$	$\frac{t_1, \sigma \mapsto t_1', \sigma' \quad \mathcal{V}(t_1', \sigma') = \perp}{t_1 \blacktriangleright e_2, \sigma \mapsto t_1' \blacktriangleright e_2, \sigma'}$
S-FILL	S-THENFAIL
$\boxtimes \beta, \sigma \mapsto \boxtimes \beta, \sigma$	$\frac{t_1, \sigma \mapsto t_1', \sigma' \quad \mathcal{V}(t_1', \sigma') = v_1 \quad e_2 v_1, \sigma' \downarrow t_2, \sigma'' \quad \mathcal{F}(t_2, \sigma'')}{t_1 \blacktriangleright e_2, \sigma \mapsto t_1' \blacktriangleright e_2, \sigma'}$
S-UPDATE	S-THENCONT
$\blacksquare l, \sigma \mapsto \blacksquare l, \sigma$	$\frac{t_1, \sigma \mapsto t_1', \sigma' \quad \mathcal{V}(t_1', \sigma') = v_1 \quad e_2 v_1, \sigma' \downarrow t_2, \sigma'' \quad \neg \mathcal{F}(t_2, \sigma'')}{t_1 \blacktriangleright e_2, \sigma \mapsto t_2, \sigma''}$
S-FAIL	S-ORLEFT
$\not\downarrow, \sigma \mapsto \not\downarrow, \sigma$	$\frac{t_1, \sigma \mapsto t_1', \sigma' \quad \mathcal{V}(t_1', \sigma') = v_1}{t_1 \blacklozenge t_2, \sigma \mapsto t_1', \sigma'}$
S-XOR	S-ORRIGHT
$e_1 \diamond e_2, \sigma \mapsto e_1 \diamond e_2, \sigma$	$\frac{t_1, \sigma \mapsto t_1', \sigma' \quad \mathcal{V}(t_1', \sigma') = \perp \quad t_2, \sigma' \mapsto t_2', \sigma'' \quad \mathcal{V}(t_2', \sigma'') = v_2}{t_1 \blacklozenge t_2, \sigma \mapsto t_2', \sigma''}$
	S-ORNONE
	$\frac{t_1, \sigma \mapsto t_1', \sigma' \quad \mathcal{V}(t_1', \sigma') = \perp \quad t_2, \sigma' \mapsto t_2', \sigma'' \quad \mathcal{V}(t_2', \sigma'') = \perp}{t_1 \blacklozenge t_2, \sigma \mapsto t_1' \blacklozenge t_2', \sigma''}$
S-NEXT	S-AND
$t_1 \triangleright e_2, \sigma \mapsto t_1' \triangleright e_2, \sigma'$	$\frac{t_1, \sigma \mapsto t_1', \sigma' \quad t_2, \sigma' \mapsto t_2', \sigma''}{t_1 \bowtie t_2, \sigma \mapsto t_1' \bowtie t_2', \sigma''}$

FIGURE 5.12: Striding semantics

$$\boxed{t, \sigma \xRightarrow{i} t', \sigma'} \quad \text{I-HANDLE} \quad \frac{t, \sigma \xrightarrow{i} t', \sigma' \quad t', \sigma' \Downarrow t'', \sigma''}{t, \sigma \xRightarrow{i} t'', \sigma''}$$

FIGURE 5.13: Interaction semantics

Principles of stepping Considering the expression $t_1 \blacktriangleright e$, stepping away from task t_1 can only be performed when t_1 has a value: $\mathcal{V}(t_1) = v_1$. Only then can a new task t_2 be calculated from the application of the result value v_1 to the expression e . On top of that, t_2 must not be failing: $\neg \mathcal{F}(t_2)$. These principles lead to the stepping rules in Fig. 5.12. S-THENSTAY does nothing, because the left side does not have a value. S-THENFAIL covers the case that the left side has a value but the calculated successor task is failing. This rule uses the semantics of the host language to evaluate the application $e_2 v_1$. When all required conditions are fulfilled, S-THENCONT allows stepping to the successor task.

Principles of choosing Choosing between two tasks t_1 and t_2 can only be done when at least one of them has a value: $\mathcal{V}(t_1) = v_1 \vee \mathcal{V}(t_2) = v_2$. When both have a value, the left task is chosen. When none has a value, none can be chosen. These principles lead to the rules S-ORLEFT, S-ORRIGHT, and S-ORNONE, which encode that the choice operator picks the leftmost task that has a value.

5.4.4 Handling user inputs

The handling semantics is the outermost layer of the stack of semantics. It is responsible for performing external steps and choices, and for changing the values of editors. The rules of the interaction semantics are given in Fig. 5.13. The semantics is only applicable to normalised tasks t . Sending an input event i to a task t , denoted as $t, \sigma \xRightarrow{i} t', \sigma'$, first handles the event and then prepares the resulting task for the next input by normalising it.

Inputs i are formed according to the grammar in Fig. 5.14. F and S in an input encode the path to the task at which the input is targeted. There is a function $\mathcal{I}$ which calculates the possible input events a given task expects. It takes a normalised task and a state and returns a set of inputs that can be handled. The definition of this function is listed in Fig. 5.15.

Handling input is done by the *handling* semantics shown in Fig. 5.16. It is a small step semantics with labelled transitions. It takes a task t in a state σ and an input i , and yields a new task t' in a new state σ' .

Inputs

$i ::= a \mid F i \mid S i$ – action, pass to first, pass to second

Actions

$a ::= c \mid C \mid L \mid R$ – constant, continue, go left, go right

FIGURE 5.14: Input grammar

$$\begin{aligned}
 \mathcal{I} &: \text{Task} \times \text{State} \rightarrow \mathcal{P}(\text{Input}) \\
 \mathcal{I}(\Box v, \sigma) &= \{c \mid c : \beta\} \quad \textbf{where } \Box v : \text{Task } \beta \\
 \mathcal{I}(\boxtimes \beta, \sigma) &= \{c \mid c : \beta\} \\
 \mathcal{I}(\blacksquare l, \sigma) &= \{c \mid c : \beta\} \quad \textbf{where } \blacksquare l : \text{Task } \beta \\
 \mathcal{I}(\downarrow, \sigma) &= \emptyset \\
 \mathcal{I}(t_1 \blacktriangleright e_2, \sigma) &= \mathcal{I}(t_1, \sigma) \\
 \mathcal{I}(t_1 \triangleright e_2, \sigma) &= \mathcal{I}(t_1, \sigma) \\
 &\quad \cup \{C \mid \mathcal{V}(t_1, \sigma) = v_1 \wedge e_2 v_1, \sigma \Downarrow t_2, \sigma' \wedge \neg \mathcal{F}(t_2, \sigma')\} \\
 \mathcal{I}(t_1 \bowtie t_2, \sigma) &= \{F i \mid i \in \mathcal{I}(t_1, \sigma)\} \cup \{S i \mid i \in \mathcal{I}(t_2, \sigma)\} \\
 \mathcal{I}(t_1 \blacklozenge t_2, \sigma) &= \{F i \mid i \in \mathcal{I}(t_1, \sigma)\} \cup \{S i \mid i \in \mathcal{I}(t_2, \sigma)\} \\
 \mathcal{I}(e_1 \diamond e_2, \sigma) &= \{L \mid e_1, \sigma \Downarrow t_1, \sigma' \wedge \neg \mathcal{F}(t_1, \sigma')\} \cup \\
 &\quad \{R \mid e_2, \sigma \Downarrow t_2, \sigma' \wedge \neg \mathcal{F}(t_2, \sigma')\}
 \end{aligned}$$

FIGURE 5.15: Inputs

$$\boxed{t, \sigma \xrightarrow{i} t', \sigma'}$$

Editing

$$\begin{array}{ccc} \text{H-CHANGE} & \text{H-FILL} & \text{H-UPDATE} \\ \frac{v, v' : \beta}{\square v, \sigma \xrightarrow{v'} \square v', \sigma} & \frac{v : \beta}{\boxtimes \beta, \sigma \xrightarrow{v} \square v, \sigma} & \frac{\sigma(l), v : \beta}{\blacksquare l, \sigma \xrightarrow{v} \blacksquare l, \sigma[l \mapsto v]} \end{array}$$

Continuing

$$\begin{array}{ccc} \text{H-NEXT} & & \\ \frac{e_2 v_1, \sigma \Downarrow t_2, \sigma' \quad \mathcal{V}(t_1, \sigma) = v_1 \wedge \neg \mathcal{F}(t_2, \sigma')}{t_1 \triangleright e_2, \sigma \xrightarrow{C} t_2, \sigma'} & & \\ \text{H-PICKLEFT} & & \text{H-PICKRIGHT} \\ \frac{e_1, \sigma \Downarrow t_1, \sigma' \quad \neg \mathcal{F}(t_1, \sigma')}{e_1 \diamond e_2, \sigma \xrightarrow{L} t_1, \sigma'} & & \frac{e_2, \sigma \Downarrow t_2, \sigma' \quad \neg \mathcal{F}(t_2, \sigma')}{e_1 \diamond e_2, \sigma \xrightarrow{R} t_2, \sigma'} \end{array}$$

Passing

$$\begin{array}{ccc} \text{H-PASSTHEN} & & \text{H-PASSNEXT} \\ \frac{t_1, \sigma \xrightarrow{i} t'_1, \sigma'}{t_1 \blacktriangleright e_2, \sigma \xrightarrow{i} t'_1 \blacktriangleright e_2, \sigma'} & & \frac{t_1, \sigma \xrightarrow{i} t'_1, \sigma'}{t_1 \triangleright e_2, \sigma \xrightarrow{i} t'_1 \triangleright e_2, \sigma'} \\ \text{H-FIRSTAND} & & \text{H-SECONDAND} \\ \frac{t_1, \sigma \xrightarrow{i} t'_1, \sigma'}{t_1 \bowtie t_2, \sigma \xrightarrow{Fi} t'_1 \bowtie t_2, \sigma'} & & \frac{t_2, \sigma \xrightarrow{i} t'_2, \sigma'}{t_1 \bowtie t_2, \sigma \xrightarrow{Si} t_1 \bowtie t'_2, \sigma'} \\ \text{H-FIRSTOR} & & \text{H-SECONDOR} \\ \frac{t_1, \sigma \xrightarrow{i} t'_1, \sigma'}{t_1 \blacklozenge t_2, \sigma \xrightarrow{Fi} t'_1 \blacklozenge t_2, \sigma'} & & \frac{t_2, \sigma \xrightarrow{i} t'_2, \sigma'}{t_1 \blacklozenge t_2, \sigma \xrightarrow{Si} t_1 \blacklozenge t'_2, \sigma'} \end{array}$$

FIGURE 5.16: Handling semantics

The rules H-CHANGE, H-FILL, H-UPDATE describe how input events v are used to change the value of editors. Editors only accept values of the correct type.

H-NEXT handles the C(ontinue) action, that triggers an external step. As with internal stepping, this is only possible if the left side has a value and the continuation is not failing.

H-PICKLEFT, H-PICKRIGHT handle L and R inputs, that are used to pick the left or right option of an external choice.

H-PASSTHEN, H-PASSNEXT pass all events other than the continue event C to the left side.

H-FIRSTAND, H-SECONDAND, H-FIRSTOR, H-SECONDOR direct the inputs F(irst) and S(econd) to the correct branch of parallel combinations.

5.4.5 Implementation

The semantics have been implemented in the Haskell programming language (Marlow, 2010). We use data types as monads (Jaskelioff, Ghani, & Hutton, 2011), data types à la carte (Swierstra, 2008), and monad transformers for layered semantics (Peyton Jones, 2001). The source code can be found on GitHub.¹ A command-line interface is part of this implementation. It prompts users to type input events, which are parsed and processed by the interaction semantics.

Also, we constructed an implementation of $\widehat{\text{TOP}}$ combinators on top of iTasks, so that $\widehat{\text{TOP}}$ specifications can be compiled to runnable applications. This shows that $\widehat{\text{TOP}}$ is a subset of iTasks. The source code for this implementation can also be found on GitHub.²

5.5 Properties

To show our semantics are reasonable, we show that our evaluation, normalisation and handling semantics is type preserving. We additionally prove a progress theorem for our small-step handling semantics. We show that our failing function $\mathcal{F}$ indeed only indicates expressions that cannot be normalised and that allow no further interaction. Finally, we prove that the function to compute all possible inputs $\mathcal{I}$ is sound and complete.

All proofs are rather straight forward induction proofs. While it would certainly be possible to prove the theorems in this section by using a theorem

¹<https://github.com/timjs/tophat-haskell>

²<https://github.com/timjs/tophat-clean>

prover like Coq or Agda, this would be more involved than just doing the proofs by hand.

5.5.1 Type preservation

We show that the following three preservation theorems hold, Where $\Gamma, \Sigma \vdash \sigma$ means that for all $l \in \sigma$, it holds that $\Gamma, \Sigma \vdash \sigma(l) : \Sigma(l)$.

Theorem 5.5.1 (Type preservation under evaluation)

For all expressions e and states σ such that $\Gamma, \Sigma \vdash e : \tau$ and $\Gamma, \Sigma \vdash \sigma$, if $e, \sigma \Downarrow e', \sigma'$, then $\Gamma, \Sigma \vdash e' : \tau$ and $\Gamma, \Sigma \vdash \sigma'$.

Theorem 5.5.2 (Type preservation under normalisation)

For all well typed expressions e and states σ such that $\Gamma, \Sigma \vdash e : \text{TASK } \tau$ and $\Gamma, \Sigma \vdash \sigma$, if $e, \sigma \Downarrow e', \sigma'$, then $\Gamma, \Sigma \vdash e' : \text{TASK } \tau$ and $\Gamma, \Sigma \vdash \sigma'$.

Theorem 5.5.3 (Type preservation under handling)

For all well typed expressions e , states σ and inputs i such that $\Gamma, \Sigma \vdash e : \text{TASK } \tau$ and $\Gamma, \Sigma \vdash \sigma$, if $e, \sigma \xrightarrow{i} e', \sigma'$, then $\Gamma, \Sigma \vdash e' : \text{TASK } \tau$ and $\Gamma, \Sigma \vdash \sigma'$.

All three Theorems are proven to be correct by induction over e . The full proofs are listed in Appendix A. From Theorem 5.5.3 and Theorem 5.5.2 we directly obtain that the driving semantics also preserves types.

5.5.2 Progress

A well-typed term of a task type is guaranteed to progress after normalisation, unless it is failing.

We define what we mean with progress in Theorem 5.5.4.

Theorem 5.5.4 (Progress under handling)

For all well typed expressions e and states σ , if $e, \sigma \Downarrow e', \sigma'$, then either $\mathcal{F}(e', \sigma')$ or there exist e'', σ'' , and i such that $e', \sigma' \xrightarrow{i} e'', \sigma''$.

Where a well typed expression e means that $\Gamma, \Sigma \vdash e : \tau$ for some type τ , and a well typed state means that $\Gamma, \Sigma \vdash \sigma$.

If an expression e and state σ are well-typed, then after normalisation, the pair e', σ' either fails, or there exists some input i that can be handled by it under the handling semantics. To prove this Theorem, we need to show that the failing function $\mathcal{F}$ behaves as expected.

Theorem 5.5.5 (Failing means no interaction possible)

For all expressions e and states σ such that $\Gamma, \Sigma \vdash e : \text{TASK } \tau$ and $\Sigma \vdash \sigma$, and $e, \sigma \Downarrow t, \sigma'$, we have that $\mathcal{F}(t, \sigma') = \text{True}$, if and only if there is no input i such that $t, \sigma' \xrightarrow{i} t', \sigma''$ for some t' and σ'' .

The Theorem above states that an expression e and state σ are failing, if, after normalisation, there exists no input that can be handled by it. We prove the theorem by induction on t . The full proof is listed in Appendix B.

We now have the ingredients to prove Theorem 5.5.4.

Proof: Given $\Gamma, \Sigma \vdash e : \text{Task } \tau$ and $\Sigma \vdash \sigma$ and after normalisation $e, \sigma \Downarrow e', \sigma'$, we find ourselves in either one of the following situations:

There exists an i such that $e', \sigma' \xrightarrow{i} e'', \sigma''$.

There does not exist an i such that $e', \sigma' \xrightarrow{i} e'', \sigma''$. In this case, we know that $\mathcal{F}(e', \sigma')$ must be true, by Theorem 5.5.5. $\square$

5.5.3 Soundness and completeness of Inputs

To validate the function that calculates all possible inputs $\mathcal{I}$, we want to show that the set of possible inputs it produces is both sound and complete with respect to the handling semantics. By sound we mean that all inputs in the set of possible inputs can actually be handled by the handling semantics, and by complete we mean that the set of possible inputs contains all inputs that can be handled by the handling semantics. Theorem 5.5.6 expresses exactly this property.

Theorem 5.5.6 (Inputs function is sound and complete)

For all expressions e , states σ , and inputs i such that $\Gamma, \Sigma \vdash e : \tau$ and $\Sigma \vdash \sigma$, we have that $i \in \mathcal{I}(e, \sigma)$ if and only if there exists an expression e' and state σ' such that $e, \sigma \xrightarrow{i} e', \sigma'$.

We prove the above theorem by induction over a derivation of $\Gamma, \Sigma \vdash e : \text{Task } \tau$. The proof is given in Appendix C.

5.6 Related work

The work presented in this chapter lies on the boundary of many areas of study. People have looked at the problem of how to model and coordinate collaboration from many different perspectives. The following subsections give an overview of related work from the many different areas.

5.6.1 TOP implementations

iTasks As mentioned earlier, iTasks is an implementation of TOP. iTasks has many features, and its basic combinators are versatile and powerful. Simpler combinators are implemented by restricting the powerful ones. This

is useful for everyday programming, where having lots of functionality at one's fingertips is convenient and efficient. $\widehat{\text{TOP}}$ on the other hand does not include the many different variations of the step- and parallel combinators of iTasks. To name two examples, the combinators $(\gg|)$ and $(||-)$ are variations of step and parallel that ignore the value of the left task.

The task layer of iTasks is embedded in the functional programming language Clean Plasmeijer et al. (2002). iTasks uses the facilities in the host language to support recursion and higher order task programs. Since $\widehat{\text{TOP}}$ uses the simply typed λ -calculus as a host language, it does not support these features.

P. W. M. Koopman et al. (2008) and Plasmeijer et al. (2012) describe two versions of the semantics of iTasks. Both give a different semantics in the form of minimal implementations of a subset of the interface of iTasks. These semantics do not make an explicit distinction between the host language and task language and they do not provide an explicit formal semantics. We describe the $\widehat{\text{TOP}}$ semantics by giving semantic relations and an implementation, which makes our system better suited for formal reasoning (Winskel, 1993).

When comparing the features offered by $\widehat{\text{TOP}}$ to iTasks, we see that the task layer of both languages are very similar. One important distinction is the fact that iTasks has a notion of time and task stability, which $\widehat{\text{TOP}}$ does not. Nevertheless, we expect that $\widehat{\text{TOP}}$ can emulate the behaviour of the task layer of iTasks, but we leave proving this as future work.

mTasks The mTasks framework (P. Koopman et al., 2018) is an implementation of TOP geared towards IOT devices. As in $\widehat{\text{TOP}}$, its basic combinators are a subset of iTasks. However, on IOT devices it is useful to continue running tasks indefinitely, which is done in mTasks using a forever combinator. This is currently not possible in $\widehat{\text{TOP}}$, since it does not support recursion. In an upcoming version of $\widehat{\text{TOP}}$ as mentioned in Section 9.3.2, a forever combinator will be added.

As for iTasks, there is currently no formal semantics for mTasks.

5.6.2 Workflow modelling

Much research has been done into workflow modelling. This work focusses on describing the collaboration between subsystems, rather than the communication between them. The systems described in the literature follow a *boxes and arrows* model of specifying workflows. Control flow, represented by arrows, usually can go unrestricted from anywhere to anywhere else

in a workflow. TOP specifies workflows declaratively, avoiding explicit specification of control flow.

Workflow patterns Workflow patterns are regarded as special kind of the design patterns in software engineering. They are recurring patterns in workflow systems, much like the combinators defined in $\widehat{\text{TOP}}$. Work by Aalst et al. (2003) defines a comprehensive list of these patterns, and examines their availability in industry workflow software. Workflow patterns are usually described in terms of control flow graphs, and no formal specification is given, which makes comparison and formal reasoning more difficult.

Workflow Nets & YAWL Workflow Nets (WFN) (Aalst, 1998) allow for the modelling and analysis of business processes. They are graphical in nature, and clearly display how every component is related to each other. A downside of WFN is that they do not facilitate higher order constructs. Also, they are often not directly executable.

A language based on WFN that is actually directly executable is YAWL (Aalst and ter Hofstede (2005)). It facilitates modelling and execution of dynamic workflows, with support for *and*, *or* and *xor* workflow patterns. As mentioned, YAWL programs consist of WFN, and are therefore programmed visually.

BPEL BPEL (OASIS, 2019) is another popular business process calculus. The standardised language allows for the specification of actions within business processes, using an XML format. The language is mainly used for coordinating web services. Two workflow patterns are supported: execution of services can be done sequentially or in parallel. On top of that, processes can be guarded by conditionals. There is no support for higher order processes however. Processes described in BPEL can be regarded as activity graphs, and they can also be rendered as such. The specified processes in BPEL are directly executable, just like YAWL.

5.6.3 Process algebras

Process algebras model concurrency. They allow for the high-level description of interaction, communication and synchronisation between different processes. Well known examples of process algebras are CSP (C. A. R. Hoare, 1985) and CCS (Milner, 1989).

Differences There are two main differences between TOP and process algebras. The first is a difference in scope. Process algebras focus on modelling the input/output behaviour of processes, by explicitly stating which actions are sent and received at certain points in the program. The goal of process algebras is formal reasoning about the interaction between processes. Typically, one wishes to prove properties such as deadlock-freedom, liveness, or adherence to a protocol specification.

The focus of TOP on the other hand is to model collaboration patterns, with the explicit goal of not having to specify how exactly subtasks communicate. The declarative specification of data dependencies between subtasks enables TOP to hide such details.

The second difference concerns internal communication. Two forms of communication between tasks exist: Passing values to continuations and sharing data. This is different from communication in process algebras, which is based on message-passing.

Similarities There are some aspects that are similar in $\widehat{\text{TOP}}$ and process algebras. Internal communication in Hoare's CSP (C. A. R. Hoare, 1985) is introduced with the concealment operator. The semantics of CSP requires that all concealed actions are handled exhaustively before any action with the environment can take place. This is somewhat similar to $\widehat{\text{TOP}}$, where all enabled internal steps must be taken until the system can react to input events again. Contrast this with Milner's CCS (Milner, 1989), where concealed actions are visible to the outside as τ -actions, and can be interleaved with external communication.

Another similarity between $\widehat{\text{TOP}}$ and process algebras, or any system with concurrency for that matter, is the need for synchronisation. Broadly speaking, concurrency means that different parts of a program can interact with the environment independently, in an interleaved manner. Synchronisation means that only some, but not all, of the possible interleavings are desirable. The semantics of the step combinators in $\widehat{\text{TOP}}$, together with the fact that internal communication happens atomically, allows for concise and intuitive synchronisation code.

5.6.4 Reactive programming

HipHop & Esterel HipHop (Berry, Nicolas, & Serrano, 2011; Berry & Serrano, 2014) is a programming language tailored to the development of synchronous reactive web systems. From a single source, both server and client applications can be generated. Programs are written in the Hop language, a Scheme dialect. Communication is based on a reactive layer

embedded in Hop. The set of HipHop reactive statements is based on those of the Esterel language (Berry & Gonthier, 1992; Boussinot & De Simone, 1991). Each reactive component starts by specifying possible input and output events. The component then proceeds as a state machine.

Input events are sent to such a machine programmatically using Hop, or are explicitly wired to events from the client. They are optionally associated with a Hop value. As Hop is a dynamic language, and HipHop uses strings to identify events, events and their possible associated values are not statically checked. Events are aggregated until the moment the machine is asked to react. The machine is executed and reacts by building a multi-set of output events. The execution of a HipHop machine is atomic. The set of inputs is not influenced by the current computations.

As with $\widehat{\text{TOP}}$, HipHop is a DSL embedded in a general purpose programming language. Another similarity is that both specifications lead to executable server and client applications from a single source. However, both HipHop and Esterel are more low level regarding their specification. Where $\widehat{\text{TOP}}$ takes tasks and collaboration as a starting point, HipHop focusses on synchronous communication and atomic execution of reactive machines.

This difference in focus shows in the way both systems define events. In HipHop programmers can define and use their own events. Inputs in $\widehat{\text{TOP}}$ are not extensible and not visible to the developer. They are a completely separate entity living on the semantic level.

Another important difference is the way in which both systems handle events. In HipHop the programmer decides when a machine should process its events. This could be just one event, or a multi-set of events that are processed simultaneously. $\widehat{\text{TOP}}$ always processes an input the moment it occurs and only handles a single event in one instance.

Functional reactive programming The Functional Reactive Programming (FRP) paradigm allows programmers to describe dynamic changes of values in a declarative way. This is done by specifying networks of values, called behaviours, that can depend on each other and on external events. Behaviours can change over time, or triggered by events. When a behaviour changes, all other behaviours that depend on it are updated automatically. The underlying implementation that takes care of the updating usually can tie input devices, like mouse and keyboard, to event streams and behaviours to output facilities, like text fields. This allows for declarative specifications of applications with user interfaces.

The idea of FRP was pioneered by Elliott and Hudak (1997). In the meantime, there are many variants and implementations, where reactive-banana (Apfelmus, 2019), FrTime (Cooper & Krishnamurthi, 2004), and Flapjax (Meyerovich et al., 2009) belong to the most well-known.

FRP and TOP are different systems with different goals. Whereas FRP expresses automatically updating data dependencies, TOP expresses collaboration patterns. TOP has no notion of time. Tasks cannot change spontaneously over time, while behaviours can. Only input events can change task values. The biggest conceptual difference between a workflow in TOP and a data network in FRP is that an event to a task only causes updates up until the next step, while an event in FRP propagates through the whole network.

That being said, there are some concepts that are similar in TOP and FRP. The *stepper* behaviour, for example, is associated with an event and yields the value of the most recent event. This is similar to editors in TOP. Furthermore, both systems can be used to declaratively program user interfaces, albeit in FRP the programmer has to construct the GUI elements manually, and connect inputs and outputs to the correct events and behaviours. In TOP graphical user interfaces are automatically derived.

5.6.5 Session types

Session types are a type discipline that can be used to check whether communicating programs conform to a certain protocol. Session types are expressions in some process calculus that describe the input/output behaviour of such programs. Session types are useful for programming languages where modules communicate with each other via messages, like CSP, π -calculus, or Go, to name a few. The only form of messages in TOP are input events which drive execution, but modules do not communicate using messages. Therefore, session types are not relevant for TOP in the sense used in the literature.

Formal reasoning about TOP programs is one of our future goals for $\widehat{\text{TOP}}$. The ideas and techniques of session types could be useful for specifying that a list of inputs of a certain form leads to desired task values. The details are a topic for future work.

Chapter 6

Symbolic TopHat

Task-Oriented Programming (TOP) is a programming paradigm that allows declarative specification of workflows. TOP is typically used in domains where functional correctness is essential, and where failure can have financial or strategic consequences. In this chapter we aim to make formal verification of software written in TOP easier. Currently, only testing is used to verify that programs behave as intended. We use symbolic execution to guarantee that no aberrant behaviour can occur. In the previous chapter we presented $\widehat{\text{TOP}}$, a formal language that implements the core aspects of TOP. In this chapter we develop a symbolic execution semantics for TopHat. Symbolic execution allows to prove that a given property holds for all possible execution paths of TopHat programs.

We show that the symbolic execution semantics is consistent with the original TopHat semantics, by proving soundness and completeness. We present an implementation of the symbolic execution semantics in Haskell. By running example programs, we validate our approach. This chapter represents a step forward in the formal verification of TOP software.

6.1 Introduction

The Task-Oriented Programming paradigm (TOP) is an abstraction over workflow specifications. The idea of TOP is to describe the work that needs to be done, in which order, by which person. From this specification, an application can be generated that helps to coordinate people and machines to execute the work. The iTasks framework (Plasmeijer et al., 2012) is an implementation of the paradigm in the functional programming language Clean. In the previous chapter, we presented the programming language $\widehat{\text{TOP}}$, to distill the core features of TOP into a language suitable for formal treatment. The usefulness of TOP has been demonstrated in several projects that used it to implement various applications. It has been used by the Netherlands Royal Navy (Jansen & Bolderheij, 2018), the Dutch Tax Office (Stutterheim et al., 2017) and the Dutch Coast Guard (Lijnse et al., 2012).

Furthermore, it can potentially be applied in domains like healthcare and Internet of Things (P. Koopman et al., 2018).

Applications in these kinds of domains are often mission critical, where programming mistakes can have severe consequences. To verify that a $\widehat{\text{TOP}}$ program behaves as intended, we would like to show that it satisfies a given property. A common way to do this is to write test cases, or to generate random input, and verify that all outcomes satisfy the property. Writing tests manually is time consuming and cumbersome. Testing interactive applications needs people to operate the application, maybe making use of a way to record and replay interactions. With this kind of testing there is no guarantee that all possible execution paths are covered.

To overcome these issues, we apply symbolic execution. Instead of executing tasks with test input, or letting a user interactively test the application, we run tasks on symbolic input. Symbolic input consists of tokens that represent any value of a certain type. When a program branches, the execution engine records the conditions over the symbolic input that lead to the different branches. These conditions can then be compared to a given predicate to check if the predicate holds under all conditions. We let an SMT solver verify these statements.

In this way we can guarantee that given predicates over the outcome of a TOP program always hold. Since iTasks is not suitable for formal reasoning, we instead apply symbolic execution to $\widehat{\text{TOP}}$ as introduced in Chapter 5 and also in (Steenvoorden, Naus, & Klinik, 2019), by systematically changing the semantic rules of the original language.

The symbolic execution system for $\widehat{\text{TOP}}$ will also power the automatic hint generation presented in Chapter 7.

6.2 Examples

In this section we study three examples to illustrate what kind of properties of task-oriented programs we would like to prove.

6.2.1 Positive value

This example demonstrates how we can prove that the first observable value of a program can only be a positive number.

Example 6.2.1

Consider the program in Listing 6.1. It asks the user to input a value of type INT. This value is then passed on to the right hand side. If the value is greater than zero, an editor containing the entered value is returned. At this point, the task has an

observable value, and we consider it done. Otherwise the step does not proceed and the task does not have an observable value. The user can enter a different input value.

$\boxtimes \text{INT} \triangleright \lambda x. \text{if } x > 0 \text{ then } \Box x \text{ else } \zeta$

LISTING 6.1: A task that only steps on a positive input value.

Imagine that we want to prove that no matter which value is given as input, the first observable value is a value greater than zero.

Symbolic execution of this program proceeds as follows. The symbolic execution engine generates a fresh symbolic input s for the editor on the left. The engine then arrives at the conditional. To take the then-branch, the condition $s > 0$ needs to hold. This branch will then result in $\Box s$, in which case the program has an observable value. The engine records this endpoint together with its path condition $s > 0$. The else-branch applies if the condition does not hold, but this leads to a failing task. Therefore, the step is not taken and the task expression is not altered. No additional program state is generated.

Symbolic execution returns a list of all possible program end states, together with the path conditions that led to them. If all end states satisfy the desired property, it is guaranteed that the property holds for all possible inputs.

In this example, the only end state is the expression $\Box s$ with path condition $s > 0$. From that we can conclude that no matter what input is given, the only result value possible is greater than zero.

6.2.2 Tax subsidy request

Stutterheim et al. (2017) worked with the Dutch tax office to develop a demonstrator for a fictional but realistic law about solar panel subsidies. In this section we study a simplified version of this, translated to $\widehat{\text{TOP}}$, to illustrate how symbolic execution can be used to prove that the program implements the law.

This example proves that a citizen will get subsidy only under the following conditions.

- The roofing company has confirmed that they installed solar panels for the citizen.
- The tax officer has approved the request.

- The tax officer can only approve the request if the roofing company has confirmed, and the request is filed within one year of the invoice date.
- The amount of the granted subsidy is at most 600 EUR.

```

let today = 13 Feb 2020 in                                1
let provideDocuments =  $\boxtimes$ Amount  $\boxtimes$   $\boxtimes$ Date in                2
let companyConfirm =  $\square$ True  $\diamond$   $\square$ False in                3
let officerApprove =  $\lambda$ invoiceDate.  $\lambda$ date.  $\lambda$ confirmed.    4
   $\square$ False  $\diamond$  if (date — invoiceDate < 365  $\wedge$  confirmed) 5
    then  $\square$ True                                           6
    else  $\frac{1}{2}$  in                                           7
provideDocuments  $\boxtimes$  companyConfirm  $\blacktriangleright$                 8
   $\lambda$ ( $\langle$ invoiceAmount, invoiceDate $\rangle$ , confirmed) .          9
officerApprove invoiceDate today confirmed  $\blacktriangleright$   $\lambda$ approved. 10
let subsidyAmount = if approved                            11
  then min 600 (invoiceAmount / 10) else 0 in              12
 $\square$ (subsidyAmount, approved, confirmed, invoiceDate, today) 13

```

LISTING 6.2: Subsidy request and approval workflow at the Dutch tax office.

Invoice amount	Invoice date	Please make a choice
400		
		Deny Confirm

FIGURE 6.1: Graphical user interface for the task in Listing 6.2.

Example 6.2.2

Listing 6.2 shows the program. To enhance readability of the example, we omit type annotations and make use of pattern matching on tuples. The program works as follows. First, the citizen has to enter their personal information (Line 2). In the original demonstrator this included the citizen service number, name, and home address. Here, we simplified the example so that the citizen only has to enter the invoice date. A date is specified using an integer representing the number of days since 1 January 2000.

In the next step (Line 8), in parallel the citizen has to provide the invoice documents of the installed solar panels, while the roofing company has to confirm that they have actually installed solar panels at the citizen's address. Once the invoice and the confirmation are there, the tax officer has to approve the request (Line 10). The officer can always decline the request, but they can only approve it if the roofing company has confirmed and the application date is within one year of the invoice date (Line 5). The result of the program is the amount of the subsidy, together with all information needed to prove the required properties (Line 13). The graphical user interface belonging to two steps in this process are shown in Fig. 6.1.

The result of the overall task is a tuple with the subsidy amount, the officer's approval, the roofing company's confirmation, the invoice amount, the invoice date, and today's date. Returning all this information allows the following predicate to be stated, which verifies the correctness of the implementation. The predicate has 5 free variables, which correspond to the returned values.

$$\psi(s, a, c, i, t) = s \geq 0 \Rightarrow c \quad (6.1)$$

$$\wedge s > 0 \Rightarrow a \quad (6.2)$$

$$\wedge a \Rightarrow (c \wedge t - i < 365) \quad (6.3)$$

$$\wedge s \leq 600 \quad (6.4)$$

$$\wedge \neg a \Rightarrow s \equiv 0 \quad (6.5)$$

The predicate ψ states that (6.1) if subsidy s has been payed, the roofing company must have confirmed c , (6.2) if subsidy has been payed, the officer must have approved a , (6.3) the officer can approve only if the roofing company has confirmed and today's date t is within 356 days of the invoice date i , and (6.4) the subsidy is maximal 600 EUR. Finally, (6.5) if the officer has not approved, the subsidy must be 0.

6.2.3 Flight booking

Recall the flight booking system from Section 5.3. We prove that when the program terminates, every passenger has exactly one seat, and that no two passengers have the same seat. The example program listed below is a simplified version of what we presented in Section 5.3.

Example 6.2.3

The program, shown in Listing 6.3, consists of three parallel seat booking tasks (Line 7). There is a shared list that stores all booked seats so far (Line 2). To book a

```

let maxSeats = 50 in                                1
let bookedSeats = ref [] in                            2
let bookSeat =  $\boxtimes$ INT  $\blacktriangleright$   $\lambda x.$                     3
  if  $\neg (x \in !\text{bookedSeats}) \wedge x \leq \text{maxSeats}$     4
    then bookedSeats :=  $x :: !\text{bookedSeats} \blacktriangleright \lambda_. \square x$  5
    else  $\frac{1}{2}$  in                                       6
bookSeat  $\bowtie$  bookSeat  $\bowtie$  bookSeat  $\blacktriangleright \lambda_.$         7
 $\square(!\text{bookedSeats})$                                 8

```

LISTING 6.3: Flight booking.

seat, a passenger has to enter a seat number (Line 3). A guard expression makes sure that only free seats can be booked (Line 4). The exclamation mark denotes dereferencing. When the guard is satisfied, the list of booked seats is updated, and the user can see his booked seat (Line 5). The main expression runs the seat booking task three times in parallel (Line 7), simulating three concurrent customers. The program returns the list of booked seats.

With the returned list, we can state the predicate to verify the correctness of the booking process.

$$\psi(l) = \text{len } l \equiv 3 \quad (6.1)$$

$$\wedge \text{uniq } l \quad (6.2)$$

The predicate specifies that all three passengers booked exactly one seat (6.1), and that all seats are unique (6.2), which means that no two passengers booked the same seat. The unary operators for list length (len) and uniqueness (uniq) are available in the predicate language. List length is a capability of SMT-LIB, while uniq is our own addition.

6.3 Language

The language presented in this section is nearly identical to the original $\widehat{\text{TOP}}$ language presented in the previous chapter. The main difference with the original grammar is the addition of symbolic values.

Symbolic execution for functional programming languages struggles with higher order features. This topic is under active study, and is not the focus of our work (Hallahan, Xue, Bland, Jhala, & Piskac, 2019; Hallahan, Xue, & Piskac., 2017). Therefore, we restrict symbols to only represent values of basic types. This restriction is of little importance in the domains we are interested in. Allowing users to enter higher order values is not useful in most workflow applications. By restricting the input grammar to first-order

values only, we ensure that no higher-order user input can be entered. Apart from input, all other higher order features are unrestricted.

The following subsections describe in detail how all elements of the $\widehat{\text{TOP}}$ language deal with the addition of symbols.

6.3.1 Expressions, values, and types

The syntax of Symbolic $\widehat{\text{TOP}}$ is listed in Fig. 6.2. Two main changes have been made with regards to the original $\widehat{\text{TOP}}$ grammar. First, symbols s have been added to the syntax of expressions. However, they are not intended to be used by programmers, similar to locations l . Instead, they are generated by the semantics as placeholders for symbolic inputs.

Symbolic expressions

$\tilde{e} ::=$	$\lambda x : \tau. \tilde{e} \mid \tilde{e}_1 \tilde{e}_2 \mid x$	– abstraction, application, variable
	$c \mid \langle \rangle \mid u \tilde{e}_1 \mid \tilde{e}_1 o \tilde{e}_2$	– constant, unit, unary, binary operation
	if $\tilde{e}_1$ then $\tilde{e}_2$ else $\tilde{e}_3$	– conditional
	$\langle \tilde{e}_1, \tilde{e}_2 \rangle \mid \text{fst } \tilde{e} \mid \text{snd } \tilde{e}$	– pair, projections
	$[]_\beta \mid \tilde{e}_1 :: \tilde{e}_2$	– nil, cons
	$\text{head } \tilde{e} \mid \text{tail } \tilde{e}$	– first element, list tail
	ref $\tilde{e} \mid !\tilde{e} \mid \tilde{e}_1 := \tilde{e}_2 \mid l$	– references, location
	$\tilde{p} \mid s$	– symbolic pretask, symbol

Symbolic Pretasks

$\tilde{p} ::=$	$\square \tilde{e} \mid \boxtimes \beta \mid \blacksquare \tilde{e}$	– valued editor, unvalued editor, shared editor
	$\tilde{e}_1 \blacktriangleright \tilde{e}_2 \mid \tilde{e}_1 \triangleright \tilde{e}_2$	– internal step, external step
	$\text{fz} \mid \tilde{e}_1 \bowtie \tilde{e}_2$	– fail, parallel composition
	$\tilde{e}_1 \blacklozenge \tilde{e}_2 \mid \tilde{e}_1 \lozenge \tilde{e}_2$	– internal choice, external choice

FIGURE 6.2: Syntax of Symbolic $\widehat{\text{TOP}}$ expressions.

Symbols are treated as values (Fig. 6.3). They have therefore been added to the grammar of values. Also, every symbol has a type, and basic operations can take symbols as arguments.

The types of Symbolic $\widehat{\text{TOP}}$ remain the same. However, we do need an additional typing rule, T-SYM in Fig. 6.4, to type symbols, since they are now part of our expression syntax. The type of symbols is kept track of in the environment Γ .

Values	
$\tilde{v} ::= \lambda x : \tau. \tilde{e} \mid \langle \tilde{v}_1, \tilde{v}_2 \rangle \mid \langle \rangle \mid c$	– abstraction, pair, unit, constant
$\mid []_\beta \mid \tilde{v}_1 :: \tilde{v}_2 \mid l \mid \tilde{t}$	– nil, cons, location, task
$\mid u \tilde{v} \mid \tilde{v}_1 o \tilde{v}_2 \mid s$	– unary/binary operation, symbol
Tasks	
$\tilde{t} ::= \square \tilde{v} \mid \boxtimes \beta \mid \blacksquare l$	– valued editor, unvalued editor, shared editor
$\mid \tilde{t}_1 \blacktriangleright \tilde{e}_2 \mid \tilde{t}_1 \triangleright \tilde{e}_2$	– internal step, external step
$\mid \downarrow \mid \tilde{t}_1 \bowtie \tilde{t}_2$	– fail, parallel combination
$\mid \tilde{t}_1 \blacklozenge \tilde{t}_2 \mid \tilde{e}_1 \diamond \tilde{e}_2$	– internal choice, external choice

FIGURE 6.3: Syntax of values in Symbolic $\widehat{\text{TOP}}$.

$$\text{T-SYM} \\ \frac{s : \beta \in \Gamma}{\Gamma, \Sigma \vdash s : \tau}$$

FIGURE 6.4: Additional typing rule for Symbolic $\widehat{\text{TOP}}$.

6.3.2 Inputs

In symbolic execution, we do not know what the input of a program will be. In our case this means that we do not know which events will be sent to editors. This is reflected in the definition of symbolic inputs and actions in Fig. 6.5

Symbolic inputs	
$\tilde{i} ::= \tilde{a} \mid F \tilde{i} \mid S \tilde{i}$	– symbolic action, to first, to second
Symbolic actions	
$\tilde{a} ::= s \mid C \mid L \mid R$	– symbol, continue, go left, go right

FIGURE 6.5: Syntax of inputs and actions in Symbolic $\widehat{\text{TOP}}$.

Inputs are still the same and consist of paths and actions. Paths are tagged with one or more F (first) and S (second) tags. Actions no longer contain concrete values, but only symbols. This means that instead of concrete values, editors can only hold symbols.

6.3.3 Path conditions

Concrete execution of $\widehat{\text{TOP}}$ programs is driven by concrete inputs, which select one branch of conditionals, or make a choice. Since no concrete information is available during symbolic execution, the symbolic execution semantics records how each execution path depends on the symbolic input. This is done by means of path conditions. Fig. 6.6 lists the syntax of path conditions.

Path conditions	
$\phi ::= c \mid s \mid \langle \rangle \mid \langle \phi_1, \phi_2 \rangle$	– constant, symbol, unit, pairs
$\mid []_\beta \mid \phi_1 :: \phi_2$	– nil, cons
$\mid u \phi \mid \phi_1 \circ \phi_2$	– unary operation, binary operation

FIGURE 6.6: Syntax of path conditions.

Path conditions are a subset of the values of basic type β . They can contain symbols, constants, pairs, lists, and operations on them.

6.4 Semantics

In this section we discuss the symbolic execution semantics for $\widehat{\text{TOP}}$. The structure of the symbolic semantics closely resembles that of the concrete semantics. It consists of three layers, a big step symbolic evaluation semantics for the host language, a big step symbolic normalisation semantics for the task language, and a small step driving semantics that processes user inputs.

They are described in the following sections. We will study their interesting aspects, and the changes made with respect to the concrete semantics.

6.4.1 Symbolic evaluation

The host language is a simply typed lambda calculus with references and basic operations. Most of the symbolic evaluation rules closely resemble the concrete semantics. The original evaluation relation ($\downarrow$) had the form $e, \sigma \downarrow v, \sigma'$, where an expression e in a state σ evaluates to a value v in state σ' . The new relation ($\Downarrow$) adds path conditions ϕ to the output and has the form $\tilde{e}, \tilde{\sigma} \Downarrow \overline{\tilde{v}, \tilde{\sigma}', \phi}$. The tilde distinguishes the symbolic variants from the concrete ones.

The symbolic semantics can generate multiple outcomes. This is denoted in the evaluation with a line over the result, which can be read as $\overline{\tilde{v}, \tilde{\sigma}', \phi} =$

$\{(\tilde{v}_1, \tilde{\sigma}'_1, \phi_1), \dots, (\tilde{v}_n, \tilde{\sigma}'_n, \phi_n)\}$. The set that results from symbolic execution can be interpreted as follows. Each element is a possible endpoint in the execution of a task. It is guarded by a condition ϕ over the symbolic input. Execution only arrives at the symbolic value $\tilde{v}$ and symbolic state $\tilde{\sigma}'$ when the path condition ϕ is satisfied.

To illustrate the difference between concrete and symbolic evaluation, Fig. 6.7 lists one rule from the concrete semantics and its corresponding symbolic counterpart.

$$\begin{array}{c}
 \text{E-EDIT} \\
 \frac{e, \sigma \downarrow v, \sigma'}{\Box e, \sigma \downarrow \Box v, \sigma'} \\
 \\
 \text{SE-EDIT} \\
 \frac{\tilde{e}, \tilde{\sigma} \Downarrow \tilde{v}, \tilde{\sigma}', \phi}{\Box \tilde{e}, \tilde{\sigma} \Downarrow \Box \tilde{v}, \tilde{\sigma}', \phi}
 \end{array}$$

FIGURE 6.7: The evaluation rule from the concrete and the symbolic semantics for the editor expression.

The E-EDIT rule evaluates the expression held in an editor to a value. The SE-EDIT does the same, but since it is concerned with symbolic execution, the expression can contain symbols. We therefore do not know beforehand which concrete value will be produced, or even which path the execution will take. If the expression contains a conditional that depends on a symbol, there can be multiple possible result values.

Most symbolic rules closely resemble their concrete counterparts, and follow directly from them.

The full symbolic evaluation semantics is listed in Fig. 6.8.

The most interesting rule is the one for conditionals. The concrete semantics has two separate rules for the **then** and the **else** branch. The symbolic semantics has one combined rule SE-IF. Since $\tilde{e}_1$ can contain symbols, it can evaluate to multiple values. The rule keeps track of all options. It calculates the **then**-branch, and records in the path condition that execution can only reach this branch if $\tilde{v}_1$ becomes True. The rule does the same for the **else**-branch, except it requires that $\tilde{v}_1$ becomes False. Note that both $\tilde{e}_2$ and $\tilde{e}_3$ are evaluated using the same state $\tilde{\sigma}'$, which is the resulting state after evaluating $\tilde{e}_1$.

6.4.2 Observations

The symbolic normalisation and driving semantics make use of observations on tasks, just like the concrete semantics.

The partial function $\mathcal{V}$ can be used to observe the value of a task. Its definition is unchanged with respect to the original.

$\boxed{\tilde{e}, \tilde{\sigma} \Downarrow \overline{\tilde{v}, \tilde{\sigma}', \phi}}$		
SE-VALUE	SE-FIRST	SE-SECOND
$\overline{\tilde{v}, \tilde{\sigma} \Downarrow \tilde{v}, \tilde{\sigma}, \text{True}}$	$\frac{\tilde{e}, \tilde{\sigma} \Downarrow \langle \tilde{v}_1, \tilde{v}_2 \rangle, \tilde{\sigma}', \phi}{\text{fst } \tilde{e}, \tilde{\sigma} \Downarrow \tilde{v}_1, \tilde{\sigma}', \phi}$	$\frac{\tilde{e}, \tilde{\sigma} \Downarrow \langle \tilde{v}_1, \tilde{v}_2 \rangle, \tilde{\sigma}', \phi}{\text{snd } \tilde{e}, \tilde{\sigma} \Downarrow \tilde{v}_2, \tilde{\sigma}', \phi}$
SE-PAIR	SE-CONS	
$\frac{\tilde{e}_1, \tilde{\sigma} \Downarrow \overline{\tilde{v}_1, \tilde{\sigma}', \phi_1} \quad \tilde{e}_2, \tilde{\sigma}' \Downarrow \overline{\tilde{v}_2, \tilde{\sigma}'', \phi_2}}{\langle \tilde{e}_1, \tilde{e}_2 \rangle, \tilde{\sigma} \Downarrow \langle \tilde{v}_1, \tilde{v}_2 \rangle, \tilde{\sigma}'', \phi_1 \wedge \phi_2}$	$\frac{\tilde{e}_1, \tilde{\sigma} \Downarrow \tilde{v}_1, \tilde{\sigma}', \phi_1 \quad \tilde{e}_2, \tilde{\sigma}' \Downarrow \tilde{v}_2, \tilde{\sigma}'', \phi_2}{\tilde{e}_1 :: \tilde{e}_2, \tilde{\sigma} \Downarrow \tilde{v}_1 :: \tilde{v}_2, \tilde{\sigma}'', \phi_1 \wedge \phi_2}$	
SE-HEAD	SE-TAIL	SE-DEREF
$\frac{\tilde{e}, \tilde{\sigma} \Downarrow \tilde{v}_1 :: \tilde{v}_2, \tilde{\sigma}', \phi}{\text{head } \tilde{e}, \tilde{\sigma} \Downarrow \tilde{v}_1, \tilde{\sigma}', \phi}$	$\frac{\tilde{e}, \tilde{\sigma} \Downarrow \tilde{v}_1 :: \tilde{v}_2, \tilde{\sigma}', \phi}{\text{tail } \tilde{e}, \tilde{\sigma} \Downarrow \tilde{v}_2, \tilde{\sigma}', \phi}$	$\frac{\tilde{e}, \tilde{\sigma} \Downarrow \overline{l, \tilde{\sigma}', \phi}}{! \tilde{e}, \tilde{\sigma} \Downarrow \tilde{\sigma}'(l), \tilde{\sigma}', \phi}$
SE-APP	SE-ASSIGN	
$\frac{\tilde{e}_1, \tilde{\sigma} \Downarrow \overline{\lambda x : \tau. \tilde{e}'_1, \tilde{\sigma}', \phi_1} \quad \tilde{e}_2, \tilde{\sigma}' \Downarrow \overline{\tilde{v}_2, \tilde{\sigma}'', \phi_2}}{\tilde{e}'_1[x \mapsto \tilde{v}_2], \tilde{\sigma}'' \Downarrow \overline{\tilde{v}_1, \tilde{\sigma}''', \phi_3}}$	$\frac{\tilde{e}_1, \tilde{\sigma} \Downarrow \overline{l, \tilde{\sigma}', \phi_1} \quad \tilde{e}_2, \tilde{\sigma}' \Downarrow \overline{\tilde{v}_2, \tilde{\sigma}'', \phi_2}}{\tilde{e}_1 := \tilde{e}_2, \tilde{\sigma} \Downarrow \langle \rangle, \tilde{\sigma}''[l \mapsto \tilde{v}_2], \phi_1 \wedge \phi_2}$	
$\tilde{e}_1 \tilde{e}_2, \tilde{\sigma} \Downarrow \overline{\tilde{v}_1, \tilde{\sigma}''', \phi_1 \wedge \phi_2 \wedge \phi_3}$		
SE-IF	SE-EDIT	
$\frac{\tilde{e}_1, \tilde{\sigma} \Downarrow \overline{\tilde{v}_1, \tilde{\sigma}', \phi_1} \quad \tilde{e}_2, \tilde{\sigma}' \Downarrow \overline{\tilde{v}_2, \tilde{\sigma}'', \phi_2} \quad \tilde{e}_3, \tilde{\sigma}' \Downarrow \overline{\tilde{v}_3, \tilde{\sigma}''', \phi_3}}{\text{if } \tilde{e}_1 \text{ then } \tilde{e}_2 \text{ else } \tilde{e}_3, \tilde{\sigma} \Downarrow \overline{\tilde{v}_2, \tilde{\sigma}'', \phi_1 \wedge \phi_2 \wedge \tilde{v}_1 \cup \tilde{v}_3, \tilde{\sigma}''', \phi_1 \wedge \phi_3 \wedge \neg \tilde{v}_1}}$	$\frac{\tilde{e}, \tilde{\sigma} \Downarrow \overline{\tilde{v}, \tilde{\sigma}', \phi}}{\Box \tilde{e}, \tilde{\sigma} \Downarrow \Box \tilde{v}, \tilde{\sigma}', \phi}$	
SE-FAIL	SE-THEN	SE-NEXT
$\frac{}{\bot, \tilde{\sigma} \Downarrow \bot, \tilde{\sigma}, \text{True}}$	$\frac{\tilde{e}_1, \tilde{\sigma} \Downarrow \overline{\tilde{t}_1, \tilde{\sigma}', \phi}}{\tilde{e}_1 \blacktriangleright \tilde{e}_2, \tilde{\sigma} \Downarrow \tilde{t}_1 \blacktriangleright \tilde{e}_2, \tilde{\sigma}', \phi}$	$\frac{\tilde{e}_1, \tilde{\sigma} \Downarrow \overline{\tilde{t}_1, \tilde{\sigma}', \phi}}{\tilde{e}_1 \triangleright \tilde{e}_2, \tilde{\sigma} \Downarrow \tilde{t}_1 \triangleright \tilde{e}_2, \tilde{\sigma}', \phi}$
SE-AND	SE-REF	
$\frac{\tilde{e}_1, \tilde{\sigma} \Downarrow \overline{\tilde{t}_1, \tilde{\sigma}', \phi_1} \quad \tilde{e}_2, \tilde{\sigma}' \Downarrow \overline{\tilde{t}_2, \tilde{\sigma}'', \phi_2}}{\tilde{e}_1 \bowtie \tilde{e}_2, \tilde{\sigma} \Downarrow \overline{\tilde{t}_1 \bowtie \tilde{t}_2, \tilde{\sigma}'', \phi_1 \wedge \phi_2}}$	$\frac{\tilde{e}, \tilde{\sigma} \Downarrow \overline{\tilde{v}, \tilde{\sigma}', \phi} \quad l \notin \text{Dom}(\sigma')}{\text{ref } \tilde{e}, \tilde{\sigma} \Downarrow \overline{l, \tilde{\sigma}'[l \mapsto \tilde{v}], \phi}}$	
SE-OR	SE-UPDATE	
$\frac{\tilde{e}_1, \tilde{\sigma} \Downarrow \overline{\tilde{t}_1, \tilde{\sigma}', \phi_1} \quad \tilde{e}_2, \tilde{\sigma}' \Downarrow \overline{\tilde{t}_2, \tilde{\sigma}'', \phi_2}}{\tilde{e}_1 \blacklozenge \tilde{e}_2, \tilde{\sigma} \Downarrow \overline{\tilde{t}_1 \blacklozenge \tilde{t}_2, \tilde{\sigma}'', \phi_1 \wedge \phi_2}}$	$\frac{\tilde{e}, \tilde{\sigma} \Downarrow \overline{l, \tilde{\sigma}', \phi}}{\blacksquare \tilde{e}, \tilde{\sigma} \Downarrow \blacksquare l, \tilde{\sigma}', \phi}$	

FIGURE 6.8: The evaluation semantics of Symbolic $\widehat{\text{TOP}}$.

$$\begin{aligned}
\mathcal{F} : \text{Task} \times \text{State} &\rightarrow \text{Bool} \\
\mathcal{F}(\square \tilde{v}, \tilde{\sigma}) &= \text{False} \\
\mathcal{F}(\boxtimes \beta, \tilde{\sigma}) &= \text{False} \\
\mathcal{F}(\blacksquare l, \tilde{\sigma}) &= \text{False} \\
\mathcal{F}(\downarrow, \tilde{\sigma}) &= \text{True} \\
\mathcal{F}(\tilde{t}_1 \blacktriangleright \tilde{e}_2, \tilde{\sigma}) &= \mathcal{F}(\tilde{t}_1, \tilde{\sigma}) \\
\mathcal{F}(\tilde{t}_1 \triangleright \tilde{e}_2, \tilde{\sigma}) &= \mathcal{F}(\tilde{t}_1, \tilde{\sigma}) \\
\mathcal{F}(\tilde{t}_1 \bowtie \tilde{t}_2, \tilde{\sigma}) &= \mathcal{F}(\tilde{t}_1, \tilde{\sigma}) \wedge \mathcal{F}(\tilde{t}_2, \tilde{\sigma}) \\
\mathcal{F}(\tilde{t}_1 \blacklozenge \tilde{t}_2, \tilde{\sigma}) &= \mathcal{F}(\tilde{t}_1, \tilde{\sigma}) \wedge \mathcal{F}(\tilde{t}_2, \tilde{\sigma}) \\
\mathcal{F}(\tilde{e}_1 \diamond \tilde{e}_2, \tilde{\sigma}) &= \bigwedge \left(\{ \mathcal{F}(\tilde{t}_1, \tilde{\sigma}'_1) \mid \tilde{e}_1, \tilde{\sigma} \Downarrow \tilde{t}_1, \tilde{\sigma}'_1 \} \cup \{ \mathcal{F}(\tilde{t}_2, \tilde{\sigma}'_2) \mid \tilde{e}_2, \tilde{\sigma} \Downarrow \tilde{t}_2, \tilde{\sigma}'_2 \} \right)
\end{aligned}$$

FIGURE 6.9: Task failing observation function $\mathcal{F}$.

The function $\mathcal{F}$ observes if a task is failing. Its definition is given in Fig. 6.9. A task is failing if it is the fail task ($\downarrow$), or if it consists of only failing tasks. This function differs from its concrete counterpart in the clause for user choice. As symbolic normalisation can yield multiple results, all of the results must be failing to make a user choice failing.

6.4.3 Normalisation

Normalization ($\Downarrow$) reduces tasks until they are ready to receive input. Very little has to be changed to accommodate symbolic execution. Just like the evaluation semantics it now gathers sets of results, each element guarded by a path condition. Fig. 6.10 lists the normalisation semantics.

$\tilde{e}, \tilde{\sigma} \Downarrow \tilde{t}, \tilde{\sigma}', \phi$ SN-DONE $\frac{\tilde{e}, \tilde{\sigma} \Downarrow \tilde{t}, \tilde{\sigma}', \phi_1}{\tilde{t}, \tilde{\sigma}' \rightsquigarrow \tilde{t}', \tilde{\sigma}'', \phi_2}$ $\frac{\tilde{\sigma}' = \tilde{\sigma}'' \wedge \tilde{t} = \tilde{t}'}{\tilde{e}, \tilde{\sigma} \Downarrow \tilde{t}, \tilde{\sigma}', \phi_1 \wedge \phi_2}$	SN-REPEAT $ \frac{\tilde{e}, \tilde{\sigma} \Downarrow \tilde{t}, \tilde{\sigma}', \phi_1}{\tilde{t}, \tilde{\sigma}' \rightsquigarrow \tilde{t}', \tilde{\sigma}'', \phi_2} $ $ \frac{\tilde{\sigma}' \neq \tilde{\sigma}'' \vee \tilde{t} \neq \tilde{t}'}{\tilde{t}', \tilde{\sigma}'' \Downarrow \tilde{t}'', \tilde{\sigma}''', \phi_3} $ $ \frac{}{\tilde{e}, \tilde{\sigma} \Downarrow \tilde{t}'', \tilde{\sigma}''', \phi_1 \wedge \phi_2 \wedge \phi_3} $
--	--

FIGURE 6.10: Symbolic normalisation semantics.

Normalisation makes use of the small step striding semantics ($\rightsquigarrow$). Fig. 6.11 lists the symbolic striding semantics.

$\boxed{\tilde{t}, \tilde{\sigma} \rightsquigarrow \overline{\tilde{t}', \tilde{\sigma}', \phi}}$		
SS-EDIT	SS-FILL	SS-UPDATE
$\square \tilde{v}, \tilde{\sigma} \rightsquigarrow \square \tilde{v}, \tilde{\sigma}, \text{True}$	$\boxtimes \beta, \tilde{\sigma} \rightsquigarrow \boxtimes \beta, \tilde{\sigma}, \text{True}$	$\blacksquare l, \tilde{\sigma} \rightsquigarrow \blacksquare l, \tilde{\sigma}, \text{True}$
SS-THENCONT	SS-THENFAIL	
$\tilde{t}_1, \tilde{\sigma} \rightsquigarrow \overline{\tilde{t}'_1, \tilde{\sigma}', \phi_1} \quad \mathcal{V}(\tilde{t}'_1, \tilde{\sigma}') = \tilde{v}_1$	$\tilde{t}_1, \tilde{\sigma} \rightsquigarrow \overline{\tilde{t}'_1, \tilde{\sigma}', \phi} \quad \mathcal{V}(\tilde{t}'_1, \tilde{\sigma}') = \tilde{v}_1$	
$\tilde{e}_2 \tilde{v}_1, \tilde{\sigma}' \Downarrow \overline{\tilde{t}_2, \tilde{\sigma}'', \phi_2} \quad \neg \mathcal{F}(\tilde{t}_2, \tilde{\sigma}'')$	$\tilde{e}_2 \tilde{v}_1, \tilde{\sigma}' \Downarrow \overline{\tilde{t}_2, \tilde{\sigma}'', -} \quad \mathcal{F}(\tilde{t}_2, \tilde{\sigma}'')$	
$\tilde{t}_1 \blacktriangleright \tilde{e}_2, \tilde{\sigma} \rightsquigarrow \overline{t_2, \sigma'', \phi_1 \wedge \phi_2}$	$\tilde{t}_1 \blacktriangleright \tilde{e}_2, \tilde{\sigma} \rightsquigarrow \overline{\tilde{t}'_1 \blacktriangleright \tilde{e}_2, \tilde{\sigma}', \phi}$	
SS-THENSTAY	SS-ORNONE	
$\tilde{t}_1, \tilde{\sigma} \rightsquigarrow \overline{\tilde{t}'_1, \tilde{\sigma}', \phi}$	$\tilde{t}_1, \tilde{\sigma} \rightsquigarrow \overline{\tilde{t}'_1, \tilde{\sigma}', \phi_1} \quad \mathcal{V}(\tilde{t}'_1, \tilde{\sigma}') = \perp$	
$\mathcal{V}(\tilde{t}'_1, \tilde{\sigma}') = \perp$	$\tilde{t}_2, \tilde{\sigma}' \rightsquigarrow \overline{\tilde{t}'_2, \tilde{\sigma}'', \phi_2} \quad \mathcal{V}(\tilde{t}'_2, \tilde{\sigma}'') = \perp$	
$\tilde{t}_1 \blacktriangleright \tilde{e}_2, \tilde{\sigma} \rightsquigarrow \overline{\tilde{t}'_1 \blacktriangleright \tilde{e}_2, \tilde{\sigma}', \phi}$	$\tilde{t}_1 \blacklozenge \tilde{t}_2, \tilde{\sigma} \rightsquigarrow \overline{\tilde{t}'_1 \blacklozenge \tilde{t}'_2, \tilde{\sigma}'', \phi_1 \wedge \phi_2}$	
SS-ORLEFT	SS-ORRIGHT	
$\tilde{t}_1, \tilde{\sigma} \rightsquigarrow \overline{\tilde{t}'_1, \tilde{\sigma}', \phi}$	$\tilde{t}_1, \tilde{\sigma} \rightsquigarrow \overline{\tilde{t}'_1, \tilde{\sigma}', \phi_1} \quad \mathcal{V}(\tilde{t}'_1, \tilde{\sigma}') = \perp$	
$\mathcal{V}(\tilde{t}'_1, \tilde{\sigma}') = \tilde{v}_1$	$\tilde{t}_2, \tilde{\sigma}' \rightsquigarrow \overline{\tilde{t}'_2, \tilde{\sigma}'', \phi_2} \quad \mathcal{V}(\tilde{t}'_2, \tilde{\sigma}'') = \tilde{v}_2$	
$\tilde{t}_1 \blacklozenge \tilde{t}_2, \tilde{\sigma} \rightsquigarrow \overline{\tilde{t}'_1, \tilde{\sigma}', \phi}$	$\tilde{t}_1 \blacklozenge \tilde{t}_2, \tilde{\sigma} \rightsquigarrow \overline{\tilde{t}'_2, \tilde{\sigma}'', \phi_1 \wedge \phi_2}$	
SS-FAIL	SS-XOR	
$\downarrow, \tilde{\sigma} \rightsquigarrow \downarrow, \tilde{\sigma}, \text{True}$	$\tilde{e}_1 \blacklozenge \tilde{e}_2, \tilde{\sigma} \rightsquigarrow \overline{\tilde{e}_1 \blacklozenge \tilde{e}_2, \tilde{\sigma}, \text{True}}$	
SS-NEXT	SS-AND	
$\tilde{t}_1, \tilde{\sigma} \rightsquigarrow \overline{\tilde{t}'_1, \tilde{\sigma}', \phi}$	$\tilde{t}_1, \tilde{\sigma} \rightsquigarrow \overline{\tilde{t}'_1, \tilde{\sigma}', \phi_1}$	
$\tilde{t}_1 \triangleright \tilde{e}_2, \tilde{\sigma} \rightsquigarrow \overline{\tilde{t}'_1 \triangleright \tilde{e}_2, \tilde{\sigma}', \phi}$	$\tilde{t}_2, \tilde{\sigma}' \rightsquigarrow \overline{\tilde{t}'_2, \tilde{\sigma}'', \phi_2}$	
	$\tilde{t}_1 \bowtie \tilde{t}_2, \tilde{\sigma} \rightsquigarrow \overline{\tilde{t}'_1 \bowtie \tilde{t}'_2, \tilde{\sigma}'', \phi_1 \wedge \phi_2}$	

FIGURE 6.11: The striding semantics of Symbolic $\widehat{\text{TOP}}$

6.4.4 Handling

The handling semantics ($\leadsto$) deals with user input. In the symbolic case there are symbols instead of concrete inputs. A complete overview of the rules can be found in Fig. 6.12.

The three rules for the editors (SH-CHANGE, SH-FILL, SH-UPDATE) clearly show how symbols enter the symbolic execution. The first one for example generates a fresh symbol s and returns an editor containing it.

There are several task combinators where the result depends on user input. For example, the parallel combinator ($\bowtie$) receives an input for either the left or the right branch. To accommodate for all possibilities, the SH-AND rule generates both cases. It tags the inputs for the first branch with F and inputs for the second branch with S .

The same principle applies to the external choice combinator ($\diamond$). The three rules SH-PICKLEFT, SH-PICKRIGHT, and SH-PICK are needed to disallow choosing failing tasks. There is one rule for the case where only the right is failing, one rule when the left is failing, and one for when none of the options are failing.

After input has been handled, tasks are normalised. The combination of those two steps is taken care of by the driving ($\rightsquigarrow$) semantics, listed in Fig. 6.14.

6.4.5 Simulating

The symbolic driving semantics is a small step semantics. Every step simulates one symbolic input. To compute every possible execution, the driving semantics needs to be applied repeatedly, until the task is done. We define a task to be done when it has an observable value: $\mathcal{V}(t', \sigma') \neq \perp$. The simulation function listed in Fig. 6.13 is recursively called to produce a list of end states and path conditions. It accumulates all symbolic inputs and returns for each possible execution the observable task value v , the path condition ϕ , and the state σ . We consider a task, state and path condition to be an end state if the task value can be observed, and the path condition is satisfiable, represented by the function $\mathcal{S}$.

The recursion terminates when one of the following conditions is met.

- $\neg \mathcal{S}(\phi)$ When the path condition cannot be satisfied, we know that all future steps will not be satisfiable either. All future steps will only add more restrictions to the path condition. No future path condition will be satisfiable, and we can therefore safely remove it.
- $\mathcal{V}(t, \sigma)$ When the current task has a value it is an end state, which we can return.

$$\boxed{\tilde{l}, \tilde{\sigma} \rightsquigarrow \tilde{l}', \tilde{\sigma}', \tilde{l}, \phi}$$

SH-CHANGE $\frac{\text{fresh } s \quad \tilde{v}, s : \beta}{\Box \tilde{v}, \tilde{\sigma} \rightsquigarrow \Box s, \tilde{\sigma}, s, \text{True}}$ SH-FILL $\frac{\text{fresh } s \quad s : \beta}{\boxtimes \beta, \tilde{\sigma} \rightsquigarrow \Box s, \tilde{\sigma}, s, \text{True}}$ SH-PASSTHEN $\frac{\tilde{l}_1, \tilde{\sigma} \rightsquigarrow \tilde{l}'_1, \tilde{\sigma}', \tilde{l}, \phi}{\tilde{l}_1 \blacktriangleright \tilde{e}_2, \tilde{\sigma} \rightsquigarrow \tilde{l}'_1 \blacktriangleright \tilde{e}_2, \tilde{\sigma}', \tilde{l}, \phi}$ SH-AND $\frac{\begin{array}{c} \tilde{l}_1, \tilde{\sigma} \rightsquigarrow \tilde{l}'_1, \tilde{\sigma}_1, \tilde{l}_1, \phi_1 \\ \tilde{l}_2, \tilde{\sigma} \rightsquigarrow \tilde{l}'_2, \tilde{\sigma}_2, \tilde{l}_2, \phi_2 \end{array}}{\tilde{l}_1 \bowtie \tilde{l}_2, \tilde{\sigma} \rightsquigarrow \begin{array}{c} \tilde{l}'_1 \bowtie \tilde{l}_2, \tilde{\sigma}_1, F \tilde{l}_1, \phi_1 \\ \cup \tilde{l}_1 \bowtie \tilde{l}'_2, \tilde{\sigma}_2, S \tilde{l}_2, \phi_2 \end{array}}$ SH-OR $\frac{\begin{array}{c} \tilde{l}_1, \tilde{\sigma} \rightsquigarrow \tilde{l}'_1, \tilde{\sigma}_1, \tilde{l}_1, \phi_1 \\ \tilde{l}_2, \tilde{\sigma} \rightsquigarrow \tilde{l}'_2, \tilde{\sigma}_2, \tilde{l}_2, \phi_2 \end{array}}{\tilde{l}_1 \blacklozenge \tilde{l}_2, \tilde{\sigma} \rightsquigarrow \begin{array}{c} \tilde{l}'_1 \blacklozenge \tilde{l}_2, \tilde{\sigma}_1, F \tilde{l}_1, \phi_1 \\ \cup \tilde{l}_1 \blacklozenge \tilde{l}'_2, \tilde{\sigma}_2, S \tilde{l}_2, \phi_2 \end{array}}$ SH-PICKLEFT $\frac{\begin{array}{c} \tilde{e}_1, \tilde{\sigma} \Downarrow \tilde{l}_1, \tilde{\sigma}_1, \phi_1 \quad \neg \mathcal{F}(\tilde{l}_1, \tilde{\sigma}_1) \\ \tilde{e}_2, \tilde{\sigma} \Downarrow \tilde{l}_2, \tilde{\sigma}_2, \phi_2 \quad \mathcal{F}(\tilde{l}_2, \tilde{\sigma}_2) \end{array}}{\tilde{e}_1 \blacklozenge \tilde{e}_2, \tilde{\sigma} \rightsquigarrow \tilde{l}_1, \tilde{\sigma}_1, L, \phi_1}$	SH-UPDATE $\frac{\text{fresh } s \quad \tilde{\sigma}(l), s : \beta}{\blacksquare l, \tilde{\sigma} \rightsquigarrow \blacksquare l, \tilde{\sigma}[l \mapsto s], s, \text{True}}$ SH-PASSNEXT $\frac{\tilde{l}_1, \tilde{\sigma} \rightsquigarrow \tilde{l}'_1, \tilde{\sigma}', \tilde{l}, \phi \quad \mathcal{V}(\tilde{l}'_1, \tilde{\sigma}') = \perp}{\tilde{l}_1 \triangleright \tilde{e}_2, \tilde{\sigma} \rightsquigarrow \tilde{l}'_1 \triangleright \tilde{e}_2, \tilde{\sigma}', \tilde{l}, \phi}$ SH-PASSNEXTFAIL $\frac{\begin{array}{c} \tilde{l}_1, \tilde{\sigma} \rightsquigarrow \tilde{l}'_1, \tilde{\sigma}_1, \tilde{l}, \phi \quad \mathcal{V}(\tilde{l}'_1, \tilde{\sigma}_1) = \tilde{v}_1 \\ \tilde{e}_2 \tilde{v}_1, \tilde{\sigma}_1 \Downarrow \tilde{l}_2, \tilde{\sigma}_2, _ \quad \mathcal{F}(\tilde{l}_2, \tilde{\sigma}_2) \end{array}}{\tilde{l}_1 \triangleright \tilde{e}_2, \tilde{\sigma} \rightsquigarrow \tilde{l}'_1 \triangleright \tilde{e}_2, \tilde{\sigma}_1, \tilde{l}, \phi}$ SH-NEXT $\frac{\begin{array}{c} \tilde{l}_1, \tilde{\sigma} \rightsquigarrow \tilde{l}'_1, \tilde{\sigma}_1, \tilde{l}, \phi_1 \quad \mathcal{V}(\tilde{l}'_1, \tilde{\sigma}_1) = \tilde{v}_1 \\ \tilde{e}_2 \tilde{v}_1, \tilde{\sigma}_1 \Downarrow \tilde{l}_2, \tilde{\sigma}_2, \phi_2 \quad \neg \mathcal{F}(\tilde{l}_2, \tilde{\sigma}_2) \end{array}}{\tilde{l}_1 \triangleright \tilde{e}_2, \tilde{\sigma} \rightsquigarrow \begin{array}{c} \tilde{l}'_1 \triangleright \tilde{e}_2, \tilde{\sigma}_1, \tilde{l}, \phi_1 \\ \cup \tilde{l}_2, \tilde{\sigma}_2, C, \phi_2 \end{array}}$ SH-PICK $\frac{\begin{array}{c} \tilde{e}_1, \tilde{\sigma} \Downarrow \tilde{l}_1, \tilde{\sigma}_1, \phi_1 \quad \neg \mathcal{F}(\tilde{l}_1, \tilde{\sigma}_1) \\ \tilde{e}_2, \tilde{\sigma} \Downarrow \tilde{l}_2, \tilde{\sigma}_2, \phi_2 \quad \neg \mathcal{F}(\tilde{l}_2, \tilde{\sigma}_2) \end{array}}{\tilde{e}_1 \blacklozenge \tilde{e}_2, \tilde{\sigma} \rightsquigarrow \tilde{l}_1, \tilde{\sigma}_1, L, \phi_1 \cup \tilde{l}_2, \tilde{\sigma}_2, R, \phi_2}$ SH-PICKRIGHT $\frac{\begin{array}{c} \tilde{e}_1, \tilde{\sigma} \Downarrow \tilde{l}_1, \tilde{\sigma}_1, \phi_1 \quad \mathcal{F}(\tilde{l}_1, \tilde{\sigma}_1) \\ \tilde{e}_2, \tilde{\sigma} \Downarrow \tilde{l}_2, \tilde{\sigma}_2, \phi_2 \quad \neg \mathcal{F}(\tilde{l}_2, \tilde{\sigma}_2) \end{array}}{\tilde{e}_1 \blacklozenge \tilde{e}_2, \tilde{\sigma} \rightsquigarrow \tilde{l}_2, \tilde{\sigma}_2, R, \phi_2}$
---	--

FIGURE 6.12: Symbolic handling semantics.

$$\begin{aligned}
& \text{simulate} : \text{Task} \times \text{State} \times [\text{Input}] \times \text{Predicate} \\
& \quad \rightarrow \mathcal{P}(\text{Value} \times [\text{Input}] \times \text{Predicate}) \\
& \text{simulate}(t, \sigma, I, \phi) = \\
& \cup \{ \text{simulate}'(\text{True}, t, t', \sigma', I \oplus [i'], \phi \wedge \phi') \mid t, \sigma \approx t', \sigma', i', \phi' \} \\
& \text{simulate}' : \text{Bool} \times \text{Task} \times \text{Task} \times \text{State} \times [\text{Input}] \times \text{Predicate} \\
& \quad \rightarrow \mathcal{P}(\text{Value} \times [\text{Input}] \times \text{Predicate}) \\
& \text{simulate}'(\text{again}, t, t', \sigma', I, \phi) \\
& \quad \mid \neg \mathcal{S}(\phi) \quad \mapsto \emptyset \\
& \quad \mid \mathcal{S}(\phi) \wedge \mathcal{V}(t', \sigma') = v \quad \mapsto \{(v, I, \phi)\} \\
& \quad \mid \mathcal{S}(\phi) \wedge \mathcal{V}(t', \sigma') = \perp \wedge t' \neq t \quad \mapsto \text{simulate}(t', \sigma', I, \phi) \\
& \quad \mid \mathcal{S}(\phi) \wedge \mathcal{V}(t', \sigma') = \perp \wedge t' = t \wedge \text{again} \\
& \mapsto \cup \{ \text{simulate}'(\text{False}, t', t'', \sigma'', I \oplus [i'], \phi \wedge \phi') \mid t', \sigma' \approx t'', \sigma'', i', \phi' \} \\
& \quad \mid \mathcal{S}(\phi) \wedge \mathcal{V}(t', \sigma') = \perp \wedge t' = t \wedge \neg \text{again} \mapsto \emptyset
\end{aligned}$$

FIGURE 6.13: Simulation function definition.

$$\begin{array}{c}
\boxed{\tilde{t}, \tilde{\sigma} \approx \bar{\tilde{t}}', \tilde{\sigma}', \tilde{t}, \phi} \\
\text{SI-HANDLE} \\
\frac{\tilde{t}, \tilde{\sigma} \rightsquigarrow \bar{\tilde{t}}', \tilde{\sigma}', \tilde{t}, \phi_1 \quad \tilde{t}', \tilde{\sigma}' \rightsquigarrow \bar{\tilde{t}}'', \tilde{\sigma}'', \phi_2}{\tilde{t}, \tilde{\sigma} \approx \bar{\tilde{t}}'', \tilde{\sigma}'', \tilde{t}, \phi_1 \wedge \phi_2}
\end{array}$$

FIGURE 6.14: Symbolic interacting semantics.

$\mathcal{V}(t', \sigma') = \perp \wedge t = t' \wedge \neg \text{again}$ When the current task does not produce a value, and it is equal to the previous task except from symbol names in editors, the *simulate* function performs one look-ahead step in case the task does proceed when a fresh symbol is entered. This one step look-ahead is encoded by the parameter *again*. When this parameter is set to False, one step look-ahead has been performed and *simulate* does not continue further. If the task has a value it is returned, otherwise the branch is pruned.

To better illustrate how the *simulate* function works, we study how it simulates Listing 6.1. Fig. 6.15 gives a schematic overview of the application of *simulate*. First, it calls the drive semantics to calculate what input the task takes. Users can enter a fresh symbol s_0 , as listed on the left. The symbolic execution then branches, since it reaches a conditional. Two cases are generated. Either $s_0 > 0$, the upper branch, or $s_0 \leq 0$, the branch to the right. In the first case, the resulting task has a value, and the symbolic execution ends returning that value and the input. In the second case, the

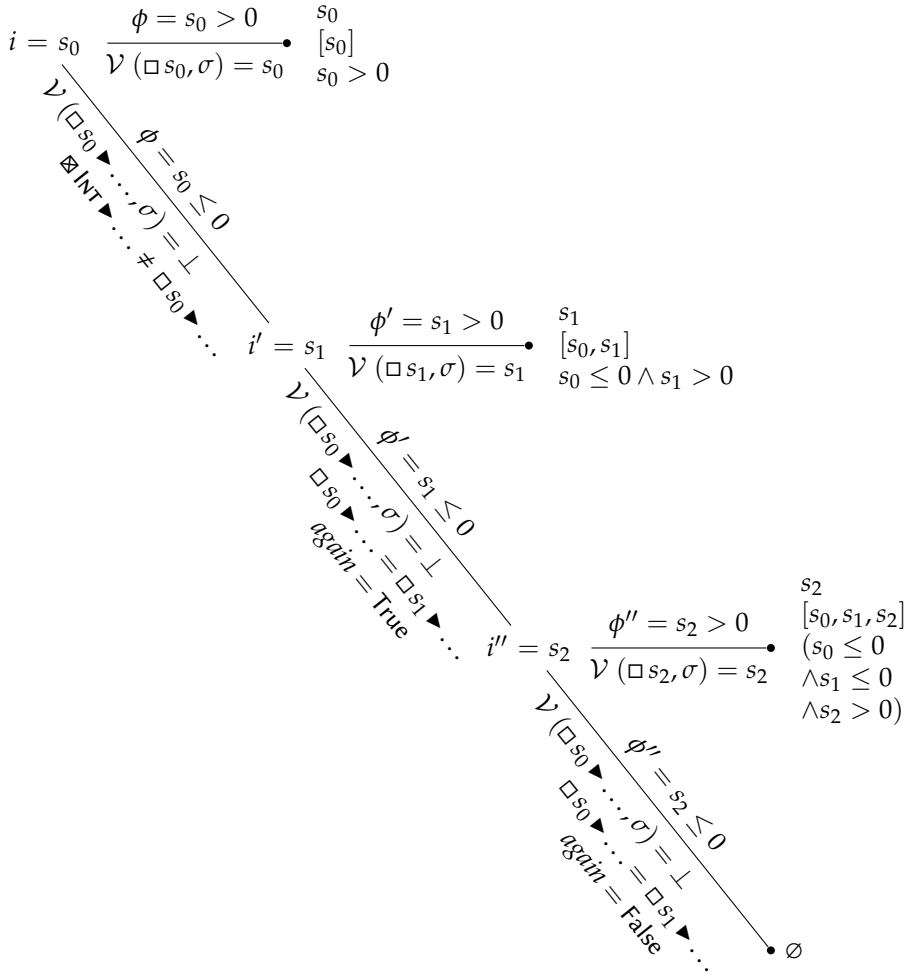


FIGURE 6.15: Application of the simulation function to Listing 6.1.

resulting task does not have a value, and the new task is different from the previous task. Therefore, it recurses, and *simulate* is called again.

A fresh symbol s_1 is generated. Again, s_1 can either be greater than zero, or less or equal. In the first case, the resulting task has a value, and the execution ends. In the second case however, the task does not have a value, and we find that the task has not been altered (apart from the new symbol). This results in a recursive call to *simulate'* with *again* set to False.

Once more a fresh symbol s_2 is generated, and s_2 can be greater than zero, or less or equal. In the first case, the task has a value and we are done. In the second case, it does not have a value, the task again has not changed, but *again* is False and therefore symbolic execution prunes this branch.

This example demonstrates a couple of things. From manual inspection, it is clear that only the first iteration returns an interesting result. When s_0 is greater than zero, the task results in a value that is greater than zero. When the input is less than or equal to zero, simulation continues with the task unchanged.

Why does the simulation still proceed then? Since the editor $\boxtimes$ changes to $\square$, the tasks are not the same after the first step. This causes *simulate* to run an extra iteration. It finds that the task still does not have a value, but now the task has changed. Then *simulate* performs one look-ahead step, by setting the *again*-parameter to False. When this look-ahead does not return a value, the branch is pruned.

6.4.6 Solving

To check the satisfiability of path conditions $\mathcal{S}(\phi)$, as well as the properties stated about a program, we make use of an external SMT solver. In the implementation we use Z3, although any other SMT solver supporting SMT-LIB could be used.

For Listing 6.1, we would like to prove that after any input sequence I , the path conditions ϕ imply that the value v of the resulting task t' is greater than 0.

$$\phi \supset v > 0 \quad \textbf{where } v = \mathcal{V}(t', \sigma')$$

As shown in Fig. 6.15, there are three paths we need to verify. Therefore, we send the following three statements to the SMT solver for verification:

1. $s_0 > 0 \Rightarrow s_0 > 0$
2. $s_0 \leq 0 \wedge s_1 > 0 \Rightarrow s_1 > 0$
3. $s_0 \leq 0 \wedge s_1 \leq 0 \wedge s_2 > 0 \Rightarrow s_2 > 0$

In this example all are trivially solvable.

6.4.7 Implementation

We implemented our language and its symbolic execution semantics in Haskell¹. With the help of a couple of GHC extensions, the grammar, typing rules and semantics are almost one-to-one translatable into code. Our tool generates execution trees like the one shown in Fig. 6.15, which keep track of intermediate normalisations, symbolic inputs, and path conditions. All path conditions are converted to SMT-LIB compatible statements and are verified using the Z3 SMT solver. As of now we do not have a parser, programs must be specified directly as abstract syntax trees.

As is usually the case with symbolic execution, the number of paths grows quickly. The examples in Listings 6.2 and 6.3 generate respectively 2112 and 1166 paths, which takes about a minute to calculate. Solving them, however, is almost instantaneous.

6.4.8 Outlook

Assertions Other work on symbolic execution often uses assertions, which are included in the program itself. One could imagine an assertion statement **assert** ψ t in $\widehat{\text{TOP}}$ that roughly works as follows. First the SMT solver verifies the property ψ against the current path condition. If the assertion fails, an error message is generated. Then the program continues with task t .

Example 6.4.1

Consider the following small example program.

$$\boxtimes \text{INT} \triangleright \lambda x. \square(\text{ref } x) \triangleright \lambda l. \text{assert } (!l \equiv x) (\square \text{"Done"})$$

This program asks the user to enter an integer. The entered value is then stored in a reference. The assertion that follows ensures that the store has been updated correctly. Finally the string "Done" is returned.

Assertions have access to all variables in scope, unlike properties as we have currently implemented them. We can overcome this by returning all values of interest at the end of the program.

$$\boxtimes \text{INT} \triangleright \lambda x. \square(\text{ref } x) \triangleright \lambda \text{store}. \square \text{"Done"} \triangleright \lambda _ . \square(x, !\text{store})$$

It is now possible to verify that the property $\psi(x, s) = x \equiv s$ holds. This demonstrates that our approach has expressive power similar to assertions. Having assertions in our language would be more convenient for programmers however, and we would like add them in the future.

¹<https://github.com/timjs/symbolic-tophat-haskell>

Input-dependent predicates Another feature we would like to support in the future are input-dependent predicates.

Example 6.4.2

Consider the following small program.

$\boxtimes_{INT} \triangleright \lambda x. \text{if } x > 0 \text{ then } \square \text{"Thank you"} \text{ else } \square \text{"Error"}$

The user inputs an integer. If the integer is larger than zero, the program prints a thank you message. If the integer is smaller than zero, an error is returned.

If we want to prove that given a positive input, the program never returns "Error", we need to be able to talk about inputs directly in predicates. Currently our symbolic execution does not support this.

6.5 Properties

In this section we describe what it means for the symbolic interaction semantics to be correct. We prove it sound and complete with respect to the concrete interaction semantics of $\widehat{\text{TOP}}$. In Chapter 7, we will also prove that these properties hold for a variant of the simulation function. We expect that soundness and completeness will also hold for the simulation definition from Fig. 6.13, but we do not prove this.

To relate the two semantics, we use the concrete inputs listed in Chapter 5.

6.5.1 Soundness

To validate the symbolic execution semantics, we want to show that for every individual symbolic execution step there exists a corresponding concrete one. This soundness property is expressed by Theorem 6.5.1.

Theorem 6.5.1 (Soundness of interact)

For all concrete tasks t , concrete states σ and mappings $M = [s_0 \mapsto c_0, \dots, s_n \mapsto c_n]$, we have for all tuples $(\tilde{t}', \tilde{\sigma}', \tilde{t}, \phi)$ in $t, \sigma \approx \tilde{t}', \tilde{\sigma}', \tilde{t}, \phi$ that $M\phi$ implies $t, \sigma \xrightarrow{M\tilde{t}} t', \sigma''$ and $M\tilde{t}' \equiv t'$ and $M\tilde{\sigma}' \equiv \sigma''$.

The proof for this theorem is rather straightforward. Since the driving semantics makes use of the handling and the normalisation semantics, we require two lemmas: one showing that the handling semantics is sound, Lemma 6.5.2, and one showing that the normalisation semantics is sound, Lemma 6.5.3.

Lemma 6.5.2 (Soundness of handling)

For all concrete tasks t , concrete states σ and mappings $M = [s_0 \mapsto c_0, \dots, s_n \mapsto c_n]$, we have for all tuples $(\tilde{t}', \tilde{\sigma}', \tilde{i}, \phi)$ in $t, \sigma \rightsquigarrow \tilde{t}', \tilde{\sigma}', \tilde{i}, \phi$, that $M\phi$ implies $t, \sigma \xrightarrow{M\tilde{i}} t', \sigma'$ and $M\tilde{t}' \equiv t'$ and $M\tilde{\sigma}' \equiv \sigma'$.

Lemma 6.5.2 is proven by induction over t . The full proof is listed in Appendix D.1.4.

Lemma 6.5.3 (Soundness of normalisation)

For all concrete expressions e , concrete states σ and mappings $M = [s_0 \mapsto c_0, \dots, s_n \mapsto c_n]$, we have for all tuples $(\tilde{t}, \tilde{\sigma}', \phi)$ in $e, \sigma \Downarrow \tilde{t}, \tilde{\sigma}', \phi$, that $M\phi$ implies $e, \sigma \Downarrow t', \sigma''$ and $M\tilde{t} \equiv t'$ and $M\tilde{\sigma}' \equiv \sigma''$.

Since Lemma 6.5.3 makes use of both the striding and the evaluation semantics, we must show soundness for those too.

Lemma 6.5.4 (Soundness of striding)

For all concrete tasks t , concrete states σ and mappings $M = [s_0 \mapsto c_0, \dots, s_n \mapsto c_n]$, we have for all tuples $(\tilde{t}', \tilde{\sigma}', \phi)$ in $t, \sigma \rightsquigarrow \tilde{t}', \tilde{\sigma}', \phi$, that $M\phi$ implies $t, \sigma \mapsto t', \sigma'$ and $M\tilde{t}' \equiv t' \wedge M\tilde{\sigma}' \equiv \sigma'$.

Lemma 6.5.5 (Soundness of evaluation)

For all concrete expressions e , concrete states σ and mappings $M = [s_0 \mapsto c_0, \dots, s_n \mapsto c_n]$, we have for all tuples $(\tilde{v}, \tilde{\sigma}', \phi)$ in $e, \sigma \Downarrow \tilde{v}, \tilde{\sigma}', \phi$, that $M\phi$ implies $e, \sigma \Downarrow v, \sigma' \wedge M\tilde{v} \equiv v \wedge M\tilde{\sigma}' \equiv \sigma'$.

The full proofs of Lemmas 6.5.3 to 6.5.5 are listed in Appendix D.1.

6.5.2 Completeness

We also want to show that for every concrete execution there exists a symbolic one.

To state this Theorem, we require a simulation relation $\tilde{i} \sim i$, which means that the symbolic input $\tilde{i}$ follows the same direction as the concrete input i . This relation is defined below.

Definition 6.5.6 (Input simulation)

A symbolic input $\tilde{i}$ simulates a concrete input i denoted as $\tilde{i} \sim i$ in the following cases.

$s \sim a$, where s is a symbol and a a concrete action.

$\tilde{i} \sim i \supset F\tilde{i} \sim Fi$

$\tilde{i} \sim i \supset S\tilde{i} \sim Si$

This allows us to define the completeness property as listed in Theorem 6.5.7.

Theorem 6.5.7 (Completeness of interact)

For all concrete tasks t , concrete states σ and concrete inputs i such that $t, \sigma \xrightarrow{i} t', \sigma'$ there exists an $\tilde{i} \sim i$, $\tilde{t}, \tilde{\sigma}$ and ϕ such that $(\tilde{t}, \tilde{\sigma}, \tilde{i}, \phi)$ in $t, \sigma \approx \tilde{t}, \tilde{\sigma}, \tilde{i}, \phi$.

The proof of Theorem 6.5.7 is rather simple. We show that handling is complete (Lemma 6.5.8) and that the subsequent normalisation is complete (Lemma 6.5.9).

Lemma 6.5.8 (Completeness of handling)

For all concrete tasks t , concrete states σ and concrete inputs i such that $t, \sigma \xrightarrow{i} t', \sigma'$ there exists an $\tilde{i} \sim i$, $\tilde{t}, \tilde{\sigma}$ and ϕ such that $(\tilde{t}, \tilde{\sigma}, \tilde{i}, \phi)$ in $t, \sigma \rightsquigarrow \tilde{t}, \tilde{\sigma}, \tilde{i}, \phi$.

Lemma 6.5.8 is proved by induction over t . We only need to show that every concrete execution is also a symbolic one. The only change needed to convert from concrete to symbolic is the adaption of the input.

Since handling makes use of normalisation and evaluation, we need to prove that they too are complete. These properties are listed in Lemmas 6.5.9 and 6.5.10

Lemma 6.5.9 (Completeness of normalisation)

For all concrete expressions e and concrete states σ such that $e, \sigma \Downarrow t, \sigma$ there exists a symbolic execution result (t, σ, True) in $e, \sigma \Downarrow \tilde{t}, \tilde{\sigma}, \phi$.

Lemma 6.5.10 (Completeness of evaluation)

For all concrete expressions e and concrete states σ such that $e, \sigma \Downarrow v, \sigma$ there exists a symbolic execution result (v, σ, True) in $e, \sigma \Downarrow \tilde{v}, \tilde{\sigma}, \phi$.

Lemmas 6.5.9 and 6.5.10 follow trivially, since every concrete execution in these semantics is also a symbolic one.

6.6 Conclusion

In this chapter, we have demonstrated how to apply symbolic execution to $\widehat{\text{TOP}}$ to verify individual programs. We have developed both a formal system and an implementation of a symbolic execution semantics. Our approach has been validated by proving the formal system correct, and by running the implementation on example programs. For these two example programs, a subsidy request workflow and a flight booking workflow, we have verified that they adhere to their specifications.

6.6.1 Future work

There are many ways in which we would like to continue this line of work.

First, we believe that more can be done with symbolic execution. Our current approach only allows proving predicates over task results and input values. We cannot, however, prove properties that depend on the order of the inputs. Since the symbolic execution currently returns a list of symbolic inputs, we think this extension is feasible.

Second, our symbolic execution only applies to $\widehat{\text{TOP}}$. We would like to see if we can fit it to iTasks. This poses several challenges. iTasks does not have a formal semantics in the sense that $\widehat{\text{TOP}}$ has. The current implementation in Clean is the closest thing available to a formal specification. There are also a few language features in iTasks that are not covered by $\widehat{\text{TOP}}$, for example loops.

Third, we would like to apply different kinds of analyses altogether. Can a certain part of the program be reached? Does a certain property hold at every point in the program? Are two programs equal? And what does it mean for two programs to be equal? We think that these properties require a different approach.

6.7 Related work

6.7.1 Symbolic execution

Symbolic execution (Boyer, Elspas, & Levitt, 1975; King, 1975) is typically being applied to imperative programming languages, for example Bucur, Kinder, and Candea (2014) prototype a symbolic execution engine for interpreted imperative languages. Cadar, Dunbar, and Engler (2008) use it to generate test cases for programs that can be compiled to LLVM bytecode. Jaffar, Murali, Navas, and Santosa (2012) use it for verifying safety properties of C programs.

In recent years it has been used for functional programming languages as well. To name some examples, there is ongoing work by Hallahan et al. (2019) to implement a symbolic execution engine for Haskell. Giantsios, Papaspyrou, and Sagonas (2017) use symbolic execution for a mix of concrete and symbolic testing of programs written in a subset of Core Erlang. Their goal is to find executions that lead to a runtime error, either due to an assertion violation or an unhandled exception. Chang, Knauth, and Torlak (2018) present a symbolic execution engine for a typed lambda calculus with mutable state where only some language constructs recognise symbolic values. They claim that their approach is easier to implement than

full symbolic execution and simplifies the burden on the solver, while still considering all execution paths.

The difficulty of symbolic execution for functional languages lies in symbolic higher-order values, that is functions as arguments to other functions. Hallahan et al. (2017) solve this with a technique called *defunctionalization*, which requires all source code to be present, so that a symbolic function can only be one of the present lambda expressions or function definitions. Giantsios et al. (2017) also require all source code to be present, but they only analyze first-order functions. They can execute higher-order functions, but only with concrete arguments. Our method also requires closed well-typed terms, so we never execute a higher-order function in isolation. Furthermore, we currently do not allow functions and tasks as task values. Together, this means that symbolic values can never be functions.

6.7.2 Contracts

Another method for guaranteeing correctness of programs are *contracts*. Contracts refine static types with additional conditions. They are enforced at runtime. Contracts were first presented by Meyer (1992) for the Eiffel programming language. Findler and Felleisen (2002) applied this technique to functional programming by implementing a contract checker for Scheme. Their contracts are assertions for higher-order programs. Contracts can be used to specify properties more fine-grained than what a static type system could check. It is possible, for example, to refine the arguments or return values of functions to numbers in a certain range, to positive numbers or non-empty lists.

Nguyen, Tobin-Hochstadt, and Horn (2017) combine contracts and symbolic execution to provide *soft contract checking*. The two ideas go hand in hand in that contracts aid symbolic execution with a language for specifications and properties for symbolic values, and symbolic execution provides compile-time guarantees and test case generation. They present a prototype implementation to verify Racket programs.

6.7.3 Axiomatic program verification

One of the classical methods of proving partial correctness of programs is Hoare's axiomatic approach (T. Hoare, 1969), which is based on pre- and postconditions. See Nielson and Nielson (1992) for a nice introduction to the topic. The axiomatic approach is usually applied to imperative programs, requires manually stating loop invariants, and manually carrying out proofs.

Some work has been done to bring the axiomatic method to functional programming. The current state of SMT solving allows for automated extraction and solving of a large amount of proof obligations. Notable works in this field are for example the Hoare Type Theory by Nanevski, Morrisett, and Birkedal (2006), the Hoare and Dijkstra Monads by Nanevski, Morrisett, Shinnar, Govereau, and Birkedal (2008); Swamy, Weinberger, Schlesinger, Chen, and Livshits (2013), or the Hoare logic for the state monad by Swierstra (2009).

The difference between the work cited here and our work is that the axiomatic method focuses on stateful computations, while we try to incorporate input as well.

Chapter 7

Assistive TopHat

Software that models business workflows is omnipresent in today's society. These systems coordinate collaboration in hospitals, companies, and military institutions. Unfortunately, workflow systems may obfuscate the influence of current user actions on the desired end result. To make the right decision, users need to oversee the full process and all information available, both of which are usually buried in the system. We have developed a way to automatically generate next-step hints for task-oriented programs. Task-oriented programming provides programmers with an abstraction over workflow software, while still being expressive enough to describe real world collaboration. By leveraging symbolic execution, we can calculate these hints without modification of the original program. To our knowledge, this is the first time that symbolic execution is used to automatically generate next-step hints for end users. We prove the generated hints to be sound and complete, and also demonstrate that the symbolic execution semantics we employ is correct for sequential input. In addition, we have developed a Haskell implementation of our automatic next-step hint generation system. By providing next-step hints, the chance of human error is reduced, while still allowing end users to intervene if required. The overall performance is raised, since the quality of decisions will improve.

7.1 Introduction

Software that supports people working together is used in most workplaces nowadays. Its aim is to automate business workflows, in order to simplify processes, to improve service, or to contain cost. In settings like hospitals, first responders and military operations, these systems could even prevent the loss of lives.

Automation and digitalisation of workflows and business processes comes at a cost. For end users it can be hard to see how an action influences their desired goal. They are unable to oversee the complete flow of the process and there might be an abundance of data that they are not fully

aware of. End users might wonder if checking a box may prevent them, or someone else, from reaching their goal, or ask themselves if they have taken all information into consideration before making a decision.

To overcome these difficulties, we propose to integrate a next-step hint system into workflow software. By combining the symbolic execution semantics for Task-Oriented Programming from the previous chapter and end-user feedback systems for rule-based problems from Chapter 2, we develop a next-step hint end-user feedback system for the Task-Oriented Programming language TopHat ($\widehat{\text{TOP}}$) (Steenvoorden et al., 2019). Our solution, which we call Assistive $\widehat{\text{TOP}}$, generates next-step hints from existing code, and does not require extra work by the programmer. To our knowledge, this is the first work employing symbolic execution to automatically generate next-step hints for end users.

Providing next-step hints to end users will provide them with a quick insight in to their situation. It reduces the chance of human error, while still allowing the user to intervene if required. The quality of decisions will improve, raising the overall performance (Power, 2002).

We start this chapter by introducing two illustrative examples. Building further on the system presented in Chapter 6, we show how we use symbolic execution to automatically generate next-step hints for end users. It is crucial that these hints are valid, meaning that they allow users to reach the desired goal. Therefore we prove correctness of the automatic hint generation system. Our hint generation system relies on symbolic execution as presented in the previous chapter. There, we proved correctness for the symbolic semantics for single user inputs. Here we prove the symbolic execution semantics to be correct for any sequence of user inputs.

7.2 Examples

This section recalls one previous example and introduces one new $\widehat{\text{TOP}}$ program. The examples will be used in Section 7.3 to demonstrate how Assistive $\widehat{\text{TOP}}$ works, and are included in the implementation.¹

7.2.1 Tax subsidy request

Recall the tax subsidy example from Section 6.2.2. We have shown that this code indeed adheres to the requirements. There we focussed on assisting the developer by proving the program correct. In this work we focus on supporting the end user that is requesting a subsidy. The end user wants

¹<https://github.com/timjs/symbolic-tophat-haskell>

```

let fork0 = ref True in                                1
let fork1 = ref True in                                2
let fork2 = ref True in                                3
let pickup =  $\lambda$ this.  $\lambda$ that.                             4
  if !this                                              5
    then  $\square$ (this := False)  $\triangleright$   $\lambda$ _.                 6
    if !that then  $\square$ (this := True) else  $\zeta$           7
  else  $\zeta$  in                                           8
let scientist =  $\lambda$ name.  $\lambda$ left.  $\lambda$ right.              9
  pickup left right  $\diamond$  pickup right left in      10
  11
  (scientist "Alan Turing" fork0 fork1  $\bowtie$            12
   scientist "Grace Hopper" fork1 fork2  $\bowtie$            13
   scientist "Ada Lovelace" fork2 fork0)              14
   $\blacktriangleright$   $\lambda$ _.  $\square$ "Full bellies"                    15

```

LISTING 7.1: Dining philosophers problem
with three computer scientists.

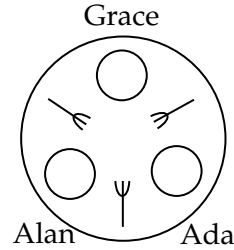


FIGURE 7.1: Rendering with
three philosophers.

the outcome of this program to be a subsidy amount larger than zero. In Section 7.3.4 we will show how to generate hints for the end user to reach this goal.

7.2.2 Dining Computer Scientists problem

The dining philosophers problem is a classic concurrency problem in computer science. A number of philosophers sit at a round table with a meal in front of them. In between the plates lies a fork. To eat their meal, each philosopher has to acquire two forks. Only after eating his or her meal, is a philosopher allowed to place the two forks back on the table. This, of course, means that the philosophers cannot eat at the same time, since there are not enough forks. Deadlock can occur when all philosophers pick up the fork to their right (or left). Then, everybody has one fork. This means that each philosopher cannot start his or her meal. Next to that, a philosopher only puts a fork back on the table after having eaten.

Example 7.2.1

We look at dining computer scientists, instead of philosophers. Listing 7.1 lists an implementation in $\widehat{\text{TOP}}$ for this problem, with three computer scientists. The forks are represented by references containing Booleans (Lines 1 to 3). Using references

allows tasks to communicate with each other across control flow. The value `True` indicates that the fork is available, `False` indicates that the fork is being used.

Picking up a fork is only possible when the fork is available, i.e. reading the reference results in `True` (Line 5). This fork is then marked as being used (Line 6). Reading a reference `l` is denoted as `!l`, assigning a new value `v` to a reference `l` is written as `l := v`.

The use of references ensures that the neighbouring scientist cannot pick up this fork: this choice will be disabled. After that, one can press `continue` if the second fork is also available (Line 7). For the sake of simplicity, the scientist sets the value of the first fork to `True`, making it available again, rather than setting the second fork to `False`, and then setting both to `True` again.

Each computer scientist takes as arguments a name and references to the two forks that he or she can reach (Line 9). They have a choice between two tasks, one that picks up the left and then the right fork, and one that first picks up the right fork followed by the left fork. This is represented with a user choice ($\diamond$, Line 10). The last lines instantiate three computer scientists sitting next to each other (Lines 12 to 14). In TOP terms, this means they collaborate in parallel ($\bowtie$) while eating their dinner, sharing some resources, in this case `fork0`, `fork1`, and `fork2`.

By design of $\widehat{\text{TOP}}$, the events of picking up a fork are performed sequentially. That is, when one computer scientist decides to pick up his right fork, we will handle that event first. After that, we will handle the choices from the other scientists. So, the order of the events is explicitly determined by the scientists themselves.

In Section 7.3.5 we will analyse this example. Our goal is to provide each scientist with a hint on which choice to make, so that they can reach the common goal of full bellies. When the scientists follow these hints, no deadlock will occur.

7.3 Generating next-step hints

This section introduces our Assistive $\widehat{\text{TOP}}$ system. The aim of Assistive $\widehat{\text{TOP}}$ is to automatically provide next-step hints. When users follow these hints, they can be sure that they will reach the goal they described beforehand. Users can, however, still decide to deviate from the given hints.

During the execution of $\widehat{\text{TOP}}$ programs, users are presented with input fields, choices and `continue` buttons. The way in which tasks progress and the resulting task value depend on these inputs. At any point during execution, we would like to present users with all possible inputs that lead users to the goal they have selected. These inputs are either concrete actions, like `continue`, `pick the left task`, `pick the right task`; or a restricted set of values to be entered into an editor. This set is restricted, since concrete

values potentially influence the flow of the program. To give a concrete example, the user should enter an integer, but this integer must be larger than zero to reach the end goal.

To come to these concrete actions and restricted values, we make use of symbolic execution. In the next two sections, we briefly describe how symbolic execution for $\widehat{\text{TOP}}$ works and recap its symbolic semantics presented Chapter 6. Thereafter, we show how to turn symbolic execution results into next-step hints. In Sections 7.3.4 and 7.3.5, we study what these automatically generated hints look like for the examples from Section 7.2.

All examples have been tested in our implementation. We added Assistive $\widehat{\text{TOP}}$ to our existing implementation of Symbolic $\widehat{\text{TOP}}^2$, which has been implemented in Haskell. It uses the Z3 SMT solver under the hood. By defining the formal hints function directly on top of the symbolic execution semantics, we can leverage the already existing symbolic execution for Symbolic $\widehat{\text{TOP}}$ in the practical implementation.

7.3.1 Symbolic execution

A symbolic execution semantics (Boyer et al., 1975; King, 1975) aims to execute a program without knowing its input. Instead, symbols are fed into the program. During evaluation, the influence of values is recorded in the path condition. The resulting symbolic value together with the path conditions can be used to prove properties of the program.

```
⊠INT ⊠ INT ▶ λ⟨x,y⟩ . if x > y then ⊠⟨y, x⟩ else ⊠⟨x, y⟩
```

LISTING 7.2: Ordering of tuple elements.

Consider the tiny example in Listing 7.2. This program asks for two integer values. After the user has entered this information, the function to the right of the step combinator makes sure the result will be an editor containing a pair, where the second element is at least as large as the first. When we run this program symbolically, we have to create fresh symbols to be entered in both editors, say s_0 and s_1 . After entering both symbolic values and then normalising the task, there are two possible outcomes, namely

- $\langle s_1, s_0 \rangle$, provided that the path condition $\phi_1 = s_0 > s_1$ holds; or
- $\langle s_0, s_1 \rangle$, with path condition $\phi_2 = \neg(s_0 > s_1)$.

²<https://github.com/timjs/symbolic-tophat-haskell>

	Concrete	Symbolic
Expressions	e	$\tilde{e}$
Tasks	t	$\tilde{t}$
States	σ	$\tilde{\sigma}$
Inputs	i	$\tilde{i}$
Evaluation	$e, \sigma \downarrow v, \sigma'$	$\tilde{e}, \tilde{\sigma} \Downarrow \tilde{v}, \tilde{\sigma}', \phi$
Normalisation	$e, \sigma \Downarrow t, \sigma'$	$\tilde{e}, \tilde{\sigma} \Downarrow \tilde{t}, \tilde{\sigma}', \phi$
Striding	$t, \sigma \mapsto t', \sigma'$	$\tilde{t}, \tilde{\sigma} \rightsquigarrow \tilde{t}', \tilde{\sigma}', \phi$
Handling	$t, \sigma \xrightarrow{i} t', \sigma'$	$\tilde{t}, \tilde{\sigma} \rightsquigarrow \tilde{t}', \tilde{\sigma}', \tilde{i}, \phi$
Interacting	$t, \sigma \xRightarrow{i} t', \sigma'$	$\tilde{t}, \tilde{\sigma} \rightsquigarrow \tilde{t}', \tilde{\sigma}', \tilde{i}, \phi$

TABLE 7.1: Overview of meta variables and semantic relations for concrete and symbolic evaluations.

Now, the property that we want to prove for this program is that no matter what the input is, the second element should always be larger than or equal to the first. We write this property as $\psi(\langle a, b \rangle) = a \leq b$. Looking at the two symbolic runs, we first need to verify that the symbolic runs are indeed viable. This is done by checking that both ϕ_1 and ϕ_2 are satisfiable, written $\mathcal{S}(\phi_1)$ and $\mathcal{S}(\phi_2)$. Symbolic runs with a path condition that is not satisfiable are discarded. Finally, we check that both path conditions conform to the goal property ψ , which is the case. Therefore, we can conclude that the property holds. When applying this technique to programs, it is a powerful tool to show that a program behaves as expected.

7.3.2 Symbolic semantics

Chapter 6 introduced the symbolic execution semantics for $\widehat{\text{TOP}}$, which we briefly repeat here.

Table 7.1 gives an overview of the entities in the concrete world, and their symbolic counterparts. Concrete expressions are a subset of symbolic expressions. Therefore, symbolic semantic relations can be applied on concrete expressions, as well as symbolic expressions.

The symbolic interaction semantics ($\rightsquigarrow$) results in a set of symbolic runs, each of them just containing one symbolic input. In other words, the symbolic interaction semantics just looks ahead one symbolic interaction. To be able to reason about an end state after multiple symbolic interactions, we introduce the notion of *simulation*. Informally, simulation performs multiple symbolic interactions after each other, until the rewritten task has an observable value. I.e. if n is the number of interactions needed to be done,

$\mathcal{V}(t'_i, \sigma'_i)$ has a result for $i = n$ but is undefined for all $i < n$. Apart from this restriction, we want to permit only viable executions. This is enforced by validating the satisfiability ($\mathcal{S}$) of the conjunction of all sequential path conditions. More formally, simulating a task for multiple user inputs is defined as follows.

Definition 7.3.1 (Simulation ($\approx^*$))

Let t and σ be a concrete task and concrete state. We define the simulation relation

$$t, \sigma \approx^* \overline{\tilde{v}, \tilde{I}, \Phi}$$

to be the set of results after performing symbolic interaction n times:

$$t, \sigma \approx \tilde{t}_1, \tilde{\sigma}_1, \tilde{t}_1, \phi_1 \approx \dots \approx \tilde{t}_n, \tilde{\sigma}_n, \tilde{t}_n, \phi_n$$

where:

- the n th task has a value: $\mathcal{V}(\tilde{t}_n, \tilde{\sigma}_n) = \tilde{v}$;
- all tasks before do not have a value: $\mathcal{V}(\tilde{t}_{i < n}, \tilde{\sigma}_{i < n}) = \perp$;
- $\tilde{I} = \tilde{t}_1 \dots \tilde{t}_n$ is the concatenation of all symbolic inputs generated along the way;
- $\Phi = \phi_1 \wedge \dots \wedge \phi_n$, is the conjunction of all path conditions encountered.

Furthermore we require that:

- the resulting predicate is satisfiable: $\mathcal{S}(\Phi)$.

The simulation definition used in this chapter differs from the one in the previous chapter. Previously, infinite symbolic executions were filtered out by allowing two steps look-ahead in case of idempotent executions. The definition above only allows finite executions by definition.

7.3.3 Next-step hints observation

As we have seen in Definition 7.3.1, a symbolic task $\tilde{t}$ is considered done as soon as it has an observable value $\tilde{v}$. To calculate next-step hints, the user needs to formulate a goal over this resulting value. The simplest goal is a predicate that always returns True, which means that a user just intends to reach the end of the task. Only when a goal has been defined, we can calculate next-step hints for end users.

Hints are calculated by means of the $\mathcal{H}$ function listed in Fig. 7.2. As input, it receives a concrete task t and concrete state σ together with a goal predicate g . The hints calculation simulates t starting in σ . This results in a set of symbolic values $\tilde{v}$, together with a list of symbolic inputs $\tilde{t} \cdot \tilde{I}$ and a

$$\mathcal{H} : \text{Task} \times \text{State} \times (\text{Value} \rightarrow \text{Bool}) \rightarrow \mathcal{P}(\text{Input} \times \text{Predicate})$$

$$\mathcal{H}(t, \sigma, g) = \{(\tilde{t}, \Phi \wedge g(\tilde{v})) \mid (t, \sigma \approx^* \tilde{v}, \tilde{t} :: \tilde{I}, \Phi), \mathcal{S}(\Phi \wedge g(\tilde{v}))\}$$

FIGURE 7.2: Definition of next-step hint function.

condition Φ to reach this path. We only want to use the symbolic executions that satisfy the goal g when applied to $\tilde{v}$. Since $\tilde{v}$ could contain symbols, it might be the case that $g(\tilde{v})$ is symbolic. It therefore forms an additional constraint on the symbolic input, and we require that the conjunction of the path condition with the goal is satisfiable ($\mathcal{S}(\Phi \wedge g(\tilde{v}))$). From the executions that fulfil this requirement, we return the first symbolic input $\tilde{t}$ from the complete list of inputs $\tilde{t} :: \tilde{I}$, together with the full condition that must hold ($\Phi \wedge g(\tilde{v})$). The resulting set contains pairs of symbolic inputs guarded by this condition.

To get a better understanding how $\mathcal{H}$ works, we study it more concretely in the next subsections. Section 7.3.4 demonstrates on the basis of the tax example briefly repeated in Section 7.2.1, how the results of symbolic execution are used to construct automatic next-step hints. Section 7.3.5 shows how hints can be generated during the execution of the example $\widehat{\text{TOP}}$ program listed in Section 7.2.2.

7.3.4 Tax subsidy request

Recall the tax example program in $\widehat{\text{TOP}}$ from Section 6.2, which models the application for a solar panel tax refund. The user enters the invoice date and invoice amount, the installation company confirms, and finally the tax officer either approves or denies the request.

In this section, we will demonstrate what symbolic execution looks like for this example, and how we generate next-step hints from the symbolic execution results. First, we call the simulate function $\approx^*$ on the program, with an empty state. The resulting set of symbolic executions is listed in Table 7.2. Each line represents one symbolic execution. In the first column, the resulting symbolic value $\tilde{v}$ is listed. The second column lists the symbolic input $\tilde{I}$ that was produced to arrive at that value, followed by the path condition Φ in the third column. The symbolic values that are produced are s_i for the invoice date and s_a for the invoice amount.

The definition of $\mathcal{H}$ describes how these results should be used to calculate next-step hints. First of all, we need a goal g to select the symbolic runs that we are interested in. The most straight forward goal would be that we want to end up in a situation where we get a subsidy amount larger than zero. This goal can be formulated as $g(\langle v, _ _ _ \rangle) = v > 0$.

Symbolic value ($\vec{v}$)	Symbolic input ($\vec{I}$)	Path condition (Φ)
$\langle \min(600, s_a/10), \text{True}, \text{True} \rangle$	$FF s_a \cdot FS s_i \cdot SL \cdot S$	$(13 \text{ Feb } 2020 - s_i) < 365 \text{ days}$
$\langle \min(600, s_a/10), \text{True}, \text{True} \rangle$	$FS s_i \cdot FF s_a \cdot SL \cdot S$	$(13 \text{ Feb } 2020 - s_i) < 365 \text{ days}$
$\langle \min(600, s_a/10), \text{True}, \text{True} \rangle$	$SL \cdot FF s_a \cdot FS s_i \cdot S$	$(13 \text{ Feb } 2020 - s_i) < 365 \text{ days}$
$\langle \min(600, s_a/10), \text{True}, \text{True} \rangle$	$SL \cdot FS s_i \cdot FF s_a \cdot S$	$(13 \text{ Feb } 2020 - s_i) < 365 \text{ days}$
$\langle \min(600, s_a/10), \text{True}, \text{True} \rangle$	$FS s_i \cdot SL \cdot FF s_a \cdot S$	$(13 \text{ Feb } 2020 - s_i) < 365 \text{ days}$
$\langle \min(600, s_a/10), \text{True}, \text{True} \rangle$	$FF s_a \cdot SL \cdot FS s_i \cdot S$	$(13 \text{ Feb } 2020 - s_i) < 365 \text{ days}$
$\langle 0, \text{False}, \text{True} \rangle$	$FF s_a \cdot FS s_i \cdot SL \cdot F$	True
$\langle 0, \text{False}, \text{True} \rangle$	$FS s_i \cdot FF s_a \cdot SL \cdot F$	True
$\langle 0, \text{False}, \text{True} \rangle$	$SL \cdot FF s_a \cdot FS s_i \cdot F$	True
$\langle 0, \text{False}, \text{True} \rangle$	$SL \cdot FS s_i \cdot FF s_a \cdot F$	True
$\langle 0, \text{False}, \text{True} \rangle$	$FS s_i \cdot SL \cdot FF s_a \cdot F$	True
$\langle 0, \text{False}, \text{True} \rangle$	$FF s_a \cdot SL \cdot FS s_i \cdot F$	True
$\langle 0, \text{False}, \text{False} \rangle$	$FF s_a \cdot FS s_i \cdot S \cdot F$	True
$\langle 0, \text{False}, \text{False} \rangle$	$FS s_i \cdot FF s_a \cdot S \cdot F$	True
$\langle 0, \text{False}, \text{False} \rangle$	$SS \cdot FF s_a \cdot FS s_i \cdot F$	True
$\langle 0, \text{False}, \text{False} \rangle$	$S \cdot FS s_i \cdot FF s_a \cdot F$	True
$\langle 0, \text{False}, \text{False} \rangle$	$FS s_i \cdot S \cdot FF s_a \cdot F$	True
$\langle 0, \text{False}, \text{False} \rangle$	$FF s_a \cdot S \cdot FS s_i \cdot F$	True

TABLE 7.2: The results of simulating the program from Listing 6.2. Only the first three elements of the symbolic value $\vec{v}$ are shown for space reasons.

The first six symbolic runs listed in Table 7.2 fulfil this goal condition. From those runs, we then take the first symbolic input, together with the path condition conjugated with the goal. After removing duplicates and redundant information, the result of $\mathcal{H}$ is as follows.

$$\begin{aligned}
 &\langle FF s_a \quad , \quad \min(600, s_a/10) > 0 \rangle \\
 &\langle FS s_i \quad , \quad (13 \text{ Feb } 2020 - s_i) < 365 \text{ days} \rangle \\
 &\langle SL \quad , \quad \text{True} \rangle
 \end{aligned}$$

This means that, at this stage, users have three possible options.³

1. The applicant may enter an amount s_a for which $\min(600, s_a/10) > 0$ should hold.
2. The applicant may enter an invoice date s_i for which $(13 \text{ Feb } 2020 - s_i) < 365 \text{ days}$ should hold.
3. The company should take the left choice (L) to confirm they installed the solar panels.

7.3.5 Dining Computer Scientists

Recall the example program Dining Computer Scientists from Section 7.2.2. Three computer scientists sit at a table and have to coordinate their eating.

³ Note that the first branch, entering an amount, is denoted by FF; the second branch, entering the invoice date, is denoted by FS; and the third branch, making a left/right choice, is denoted by S.

```

t = (scientist "Alan Turing" fork0 fork1 ⋈
  □⟨⟩ ▷ λ_.
    if !fork1 then fork2 := True else ⚡ ⋈
  □⟨⟩ ▷ λ_.
    if !fork2 then fork0 := True else ⚡)
  ▶ λ_. □ "Full bellies"
σ = {fork0 ↦ False, fork1 ↦ True, fork2 ↦ False}

```

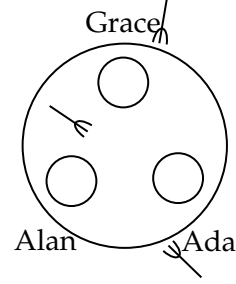


FIGURE 7.3: Task, state and visual representation of dining computer scientists after two moves.

We want to calculate all possible next steps that lead to the goal. The goal in this example is for all computer scientists to finish their meal. In terms of the resulting task value, this means that we want to reach the value "Full bellies". Witten as a predicate, we get $g(v) = v \equiv \text{"Full bellies"}$.

Let us assume that both Grace Hopper and Ada Lovelace have already picked up the forks to their left (fork2 and fork0 respectively). We then find ourselves in the situation shown in Fig. 7.3.

Calling $\mathcal{H}(t, \sigma, g)$ will result in just one hint, namely

$$\langle \text{SFC}, \text{True} \rangle$$

This means that the only step towards goal g is for the second scientist,⁴ which is Grace Hopper, to pick up the right fork. Although it is also possible for Alan Turing to pick up the fork to his left, this step is not a valid hint and performing this action will result in deadlock.

7.4 Properties

In this section, we want to validate our approach by proving correctness of Assistive $\widehat{\text{TOP}}$. For the hints function, which forms the heart of Assistive $\widehat{\text{TOP}}$, we want to prove that its results are both sound and complete. Since the hints function relies on Symbolic $\widehat{\text{TOP}}$, and more specifically, the updated definition of the simulate relation, we first prove correctness of simulate.

⁴ The third branch is denoted by SF. The action C means pushing the continue button.

7.4.1 Correctness of simulate

The symbolic execution semantics is correct when all symbolic runs relate to a concrete run, and the other way around, when all concrete runs are contained in the set of all symbolic executions. These properties are, respectively, soundness and completeness.

The simulation applies symbolic interaction multiple times. To prove certain properties with respect to the concrete semantics, we need a concrete analog of simulation. Therefore, we define *execution*, which applies concrete interaction multiple times.

Definition 7.4.1 (Execution ($\Rightarrow^*$))

Let t be a concrete task, σ a concrete state, and $I = i_1 \cdots i_n$ a list of n concrete inputs. We define the execution relation

$$t, \sigma \xRightarrow{I^*} v$$

to be the value of task t after performing concrete interaction for each input i in I :

$$t, \sigma \xRightarrow{i_1} t_1, \sigma_1 \xRightarrow{i_2} \cdots \xRightarrow{i_n} t_n, \sigma_n$$

where

- v is the value of t_n : $\mathcal{V}(t_n, \sigma_n) = v$; and
- all tasks before t_n do not have a value: $\mathcal{V}(t_{i < n}, \sigma_{i < n}) = \perp$.

We also require the notion of input simulation and the functions *SymOf* and *ValOf*, which is defined as follows.

Definition 7.4.2 (Input simulation)

A symbolic input $\tilde{i}$ simulates a concrete input i denoted as $\tilde{i} \sim i$ in the following cases.

$s \sim a$ where s is a symbol and a a concrete action.

$F \tilde{i} \sim F i$ if $\tilde{i} \sim i$

$S \tilde{i} \sim S i$ if $\tilde{i} \sim i$

$ValOf : \text{Inputs} \rightarrow \text{Values}$	$SymOf : \text{Symbolic Inputs} \rightarrow \text{Symbolic Values}$
$ValOf(F i) = ValOf(i)$	$SymOf(F i) = SymOf(i)$
$ValOf(S i) = ValOf(i)$	$SymOf(S i) = SymOf(i)$
$ValOf(c) = c$	$SymOf(s) = s$
$ValOf(_) = \perp$	$SymOf(_) = \perp$

Using execution, input simulation, and the functions *SymOf* and *ValOf*, we can state soundness and completeness for simulation as follows.

Lemma 7.4.3 (Soundness of simulate)

For all tasks t and states σ such that $t, \sigma \approx^* \overline{\tilde{v}, \tilde{I}, \Phi}$ where $\tilde{I} = \tilde{i}_0 \cdots \tilde{i}_n$, for each triple of results $\langle \tilde{v}, \tilde{I}, \Phi \rangle$ there exists a concrete input $I = i_0 \cdots i_n$ with the same length as the symbolic input $\tilde{I}$ such that $t, \sigma \xRightarrow{I}^* v$ with $[s_i \mapsto c_i] \tilde{v} = v$ and $[s_i \mapsto c_i] \Phi$ where $\text{SymOf}(\tilde{i}_i) = s_i$ and $\text{ValOf}(i_i) = c_i$.

Lemma 7.4.4 (Completeness of simulate)

For all tasks t , states σ , and lists of input $I = i_0 \cdots i_n$ such that $t, \sigma \xRightarrow{I}^* v$, there exists a symbolic value $\tilde{v}$ and a symbolic input $\tilde{I} = \tilde{i}_0 \cdots \tilde{i}_n$ with the same length as I , such that $(\tilde{v}, \tilde{I}, \Phi) \in t, \sigma \approx^* \overline{\tilde{v}, \tilde{I}, \Phi}$, with $\tilde{i}_i \sim i_i$, $[s_i \mapsto c_i] \tilde{v} = v$ and $[s_i \mapsto c_i] \Phi$, where $\text{SymOf}(\tilde{i}_i) = s_i$ and $\text{ValOf}(i_i) = c_i$.

Our strategy to prove these two lemmas is outlined in Fig. 7.4. At the top, we start out with any task t and state σ . The left side of the diagram is an overview of the execute function. Inputs i_1 until i_n are sequentially applied, until the task has an observable value.

On the right side, symbolic execution is performed. One step of the symbolic interaction semantics is taken, which results in a symbolic task, state, input and a path condition. Provided that the path condition holds, interaction is executed sequentially until the symbolic task has an observable symbolic value.

Proving soundness and completeness of simulation now comes down to relating the left and right side of the diagram. From symbolic to concrete (right to left) is soundness, as stated in Lemma 7.4.3. From concrete to symbolic (left to right) is completeness, as stated in Lemma 7.4.4.

Since simulation and execution rely on the (symbolic) handling semantics, we prove soundness and completeness of those semantics first. Looking at Fig. 7.4, there are two different settings in which the (symbolic) handling semantics are applied. At the top, both symbolic and concrete execution start out with the same task and state. But further down, the task and state differ for both semantics. The task and state are related to each other however. The symbolic semantics introduces symbols, the concrete semantics handles concrete values. This relation is expressed by the consistency relation listed in Definition 7.4.5.

Definition 7.4.5 (Consistency relation $\Leftarrow$)

A concrete task t and concrete state σ are considered to be consistent with a symbolic task $\tilde{t}$, symbolic state $\tilde{\sigma}$ and path condition Φ under a certain mapping $M = [s_1 \mapsto c_1, \dots, s_n \mapsto c_n]$, denoted as $t, \sigma \Leftarrow_M \tilde{t}, \tilde{\sigma}, \Phi$ if and only if $M\tilde{t} = t$, $M\tilde{\sigma} = \sigma$ and $M\Phi$.

Now Lemma 7.4.6 and Lemma 7.4.7 express soundness and completeness of interacting respectively.

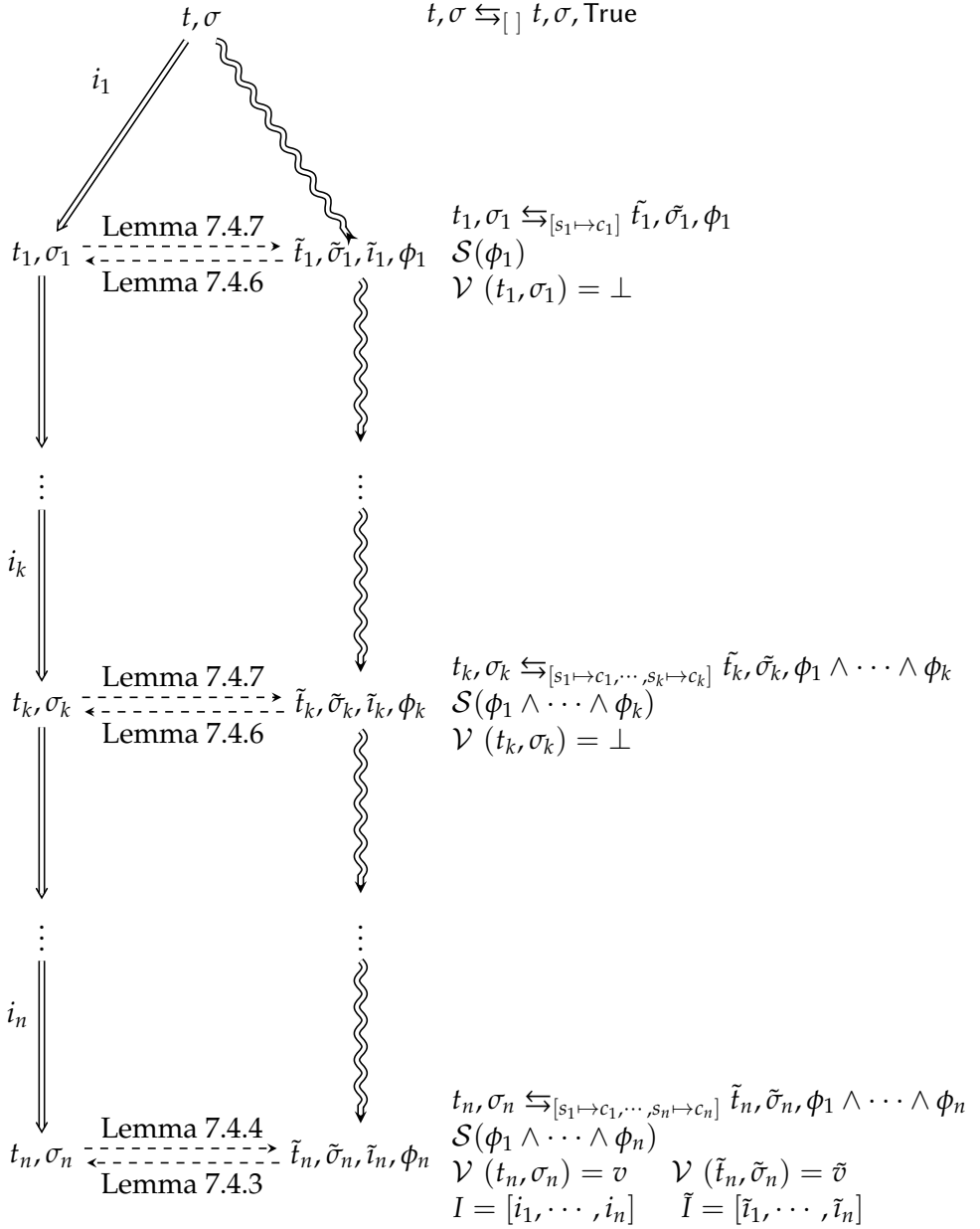


FIGURE 7.4: Proof structure

Lemma 7.4.6 (Soundness of interacting)

For all concrete tasks t , concrete states σ , symbolic tasks $\tilde{t}$, symbolic states $\tilde{\sigma}$ path conditions Φ and mappings M , we have that $t, \sigma \sqsubseteq_M \tilde{t}, \tilde{\sigma}, \Phi$ implies that for all tuples $(\tilde{t}', \tilde{\sigma}', \tilde{i}, \phi)$ in $\tilde{t}, \tilde{\sigma} \approx \tilde{t}', \tilde{\sigma}', \tilde{i}, \phi$, $\mathcal{S}(\Phi \wedge \phi)$ implies that there exists an input i such that $\tilde{i} \sim i$, $t, \sigma \xRightarrow{i} t', \sigma'$ and $t', \sigma' \sqsubseteq_{[s \mapsto c]M} \tilde{t}', \tilde{\sigma}', \Phi \wedge \phi$ where $\text{SymOf}(\tilde{i}) = s$ and $\text{ValOf}(i) = c$.

Lemma 7.4.7 (Completeness of interacting)

For all concrete tasks t , concrete states σ , symbolic tasks $\tilde{t}$, symbolic states $\tilde{\sigma}$ path conditions Φ and mappings M , we have that $t, \sigma \sqsubseteq_M \tilde{t}, \tilde{\sigma}, \Phi$ implies that for all inputs i such that $t, \sigma \xRightarrow{i} t', \sigma'$, there exists a symbolic input $\tilde{i}$, $\tilde{i} \sim i$ such that $\tilde{t}, \tilde{\sigma} \approx \tilde{t}', \tilde{\sigma}', \tilde{i}, \phi$, $\mathcal{S}(\Phi \wedge \phi)$ and $t', \sigma' \sqsubseteq_{[s \mapsto c]M} \tilde{t}', \tilde{\sigma}', \Phi \wedge \phi$ where $\text{SymOf}(\tilde{i}) = s$ and $\text{ValOf}(i) = c$.

In other words, if a symbolic and concrete task and state are related, they will still be related after (symbolic) handling. The top case, where both the symbolic and concrete semantics start out with the same task and state, can be seen as a special case of the consistency relation. Obviously a task and state are consistent with themselves, using the empty mapping and the path condition `True`.

The full proof of all four lemmas is given in Appendix E.

7.4.2 Correctness of hints

Now that soundness and completeness of simulate have been proven, we can prove that our hints function produces correct hints. Intuitively, for a next-step hint to be correct, it should adhere to the following requirements:

- it leads to concrete input users can actually insert; and
- when users follow the hint, the end goal is still reachable.

Moreover, a set of next-step hints is correct when:

- each hint it contains is correct; and
- it covers all possible inputs that lead to the end goal.

We separate these requirements into two lemmas: soundness and completeness.

Theorem 7.4.8 (Soundness of hints)

For all tasks t , states σ , and goals g , for every next-step hint $(\tilde{i}, \Phi)$ in $\mathcal{H}(t, \sigma, g)$, there exists a sequence of concrete inputs I and a concrete input i such that $\tilde{i} \sim i$, $\mathcal{S}([s \mapsto c]\Phi)$, $t, \sigma \xRightarrow{i} t', \sigma' \xRightarrow{I^*} v$ and $g(v)$, where $\text{SymOf}(\tilde{i}) = s$ and $\text{ValOf}(i) = c$.

Proof: Theorem 7.4.8 follows from the definition of $\mathcal{H}$ and Lemma 7.4.3 as follows.

The definition of $\mathcal{H}$ gives us that for every pair $(\tilde{t}, \Phi)$ produced by $\mathcal{H}$, there exists a triple $(\tilde{v}, \tilde{t} :: \tilde{is}, \Phi)$ with $\mathcal{S}(\Phi \wedge g(\tilde{v}))$. Then by Lemma 7.4.3 we have that there exists a sequence of concrete inputs I such that $t, \sigma \xRightarrow{I}^* v$ and $g(v)$. $\square$

Theorem 7.4.9 (Completeness of hints)

For all tasks t , states σ , lists of input $i \cdot I$, and goals g , if $t, \sigma \xRightarrow{I}^ v$ and $g(v)$, then there exists a symbolic input $\tilde{t}$ and path condition Φ such that $(\tilde{t}, \Phi) \in \mathcal{H}(t, \sigma, g)$ with $\tilde{t} \sim i$ and $\mathcal{S}([s \mapsto c]\Phi)$ with $\text{ValOf}(i) = c$ and $\text{SymOf}(\tilde{t}) = s$.*

Proof: To prove that i is contained in $\mathcal{H}(t, \sigma, g)$, we need to show that $t, \sigma \approx^* (\tilde{v}, \tilde{t} :: \tilde{I}, \Phi)$, with $\tilde{t} \sim i$ and $\mathcal{S}([s_0 \mapsto c_0, \dots, s_n \mapsto c_n] \wedge g(\tilde{v}))$, with $i :: I = [i_0, \dots, i_n]$, $\tilde{t} :: \tilde{I} = [\tilde{t}_0, \dots, \tilde{t}_n]$, $\text{ValOf}(i_0) = c_0, \dots, \text{ValOf}(i_n) = c_n$ and $\text{SymOf}(\tilde{t}_0) = s_0, \dots, \text{SymOf}(\tilde{t}_n) = s_n$.

By Lemma 7.4.4, we directly obtain that this indeed exists. Therefore we know that $\tilde{t}$ and Φ exist. $\square$

7.5 Related work

In the first part of this dissertation, we have attempted to provide end-users with next-step hints by viewing workflows as rule-based problems. By abstracting over workflows, reasoning about them becomes simpler. A standard search algorithm can be run to find a path to the desired goal state. The main drawback of this approach however is that a programmer needs to augment existing workflows with extra information to convert it to a rule-based problem.

Stutterheim et al. (Stutterheim et al., 2014) have developed Tonic, a task visualiser for iTasks with limited path prediction capabilities. The main goal is not to provide hints to end-users, but the system is able to handle the complete task language, and visualise the effects of user input on the progression of tasks.

To overcome the problems of our own previous research and the limited use of Tonic for end-user hints, we have combined symbolic execution, together with workflow modelling and next-step hint generation. To our knowledge, this is the first work describing the combination of these techniques in this way. The different components coming together in this chapter have been studied extensively. Section 6.7.1 gives an overview of related work on symbolic execution, Section 2.8 gives related work on workflow

modelling, and the following section presents related work on automatic hint generation in intelligent tutoring systems.

7.5.1 Automatic hint generation in intelligent tutoring systems

The intelligent tutoring systems (ITS) research community is very large. Work that is most relevant to our own is the research into automatic hint generation. More traditional ITSs rely heavily on experts to write dedicated hints for every specific case of an exercise. It is not uncommon that about 200-300 hours of development time is required per hour of instructions (Aleven, McLaren, Sewall, & Koedinger, 2009; Murray, 2003; Sottolare, Graesser, Hu, & Brawner, 2015). Automatic hint generation attempts to partly overcome this burden by calculating a hint rather than having every case specified, and thus reducing the amount of work required in the development of instructions.

Heeren et al. (Heeren & Jeuring, 2014) develop a framework for so called domain reasoners that allow for automatic hint generation. Feedback is calculated automatically from a high-level description of an exercise class. Their approach is applicable to domains like logic, mathematics and linear algebra. Paquette et al. (Paquette, Lebeau, Beaulieu, & Mayers, 2012) present a different automatic next-step hint framework, that is used to provide hints to students in a floating-point number conversion exercise.

Based on the work mentioned above by Heeren et al., an ITS for Haskell exercises has been developed by Gerdes et al. (Gerdes, Heeren, Jeuring, & van Binsbergen, 2017). It turns out that programming exercises is a popular area for automatic hint generation. Keuning et al. (Keuning, Jeuring, & Heeren, 2019) have written a literature study of this research area.

Part III

Conclusions

Chapter 8

Conclusion

Task-oriented programming (TOP) is a programming paradigm that focusses on people working together. It aims to provide a high level of abstraction over workflow systems, while still being expressive enough to model real world collaboration. It only exists in implementation, the most used system being iTasks. To prepare TOP for formal treatment, we have answered the following question.

What is the essence of the task-oriented programming paradigm?

In Chapter 4, I have presented an informal description of the essence of Task-oriented Programming (TOP). Programs written in TOP are called tasks. Tasks model collaboration, tasks are reusable, tasks are driven by user input, tasks can be observed, tasks are never done, tasks can share information and tasks are predictable.

Distilling those informal concepts into a concrete semantics, three concepts are essential: editors, combinators and shared data.

Then in Chapter 5, we formalise the informal TOP description into the programming language $\widehat{\text{TOP}}$. This is done by embedding a task layer in a simply typed lambda calculus with references, pairs and lists. A type system is given for $\widehat{\text{TOP}}$, as well as a formal semantics. By separating the host language from the task layer completely, it is clear what functionality lies where.

What makes these semantics novel is the fact that $\widehat{\text{TOP}}$ supports higher order workflows, and that it models communication with users, communication along control flow and communication across control flow. This sets it apart from other workflow modelling languages like workflow nets and BPEL that do not support higher order constructs.

For $\widehat{\text{TOP}}$, type preservation is proven. An implementation in Haskell and an implementation on top of iTasks is available.

To ensure that $\widehat{\text{TOP}}$ programs adhere to their specification, we want to prove them correct. By providing an actual proof of correctness, we are certain that programs never produce unexpected results. To do so, we answered the following question.

How can we define and guarantee properties of tasks?

We have described how to define properties over tasks. This is done by defining a predicate over the value of a task. A consequence of this definition is that we can only prove properties that must hold at the point where a task has an observable value.

To guarantee that a property holds, we have constructed a symbolic execution semantics. This semantics is able to simulate every execution. Instead of running a task on actual user input, symbols are entered in to it. The result of a task simulation is a set of pairs containing a symbolic task value, guarded by a path condition.

We have shown that when a predicate holds for every symbolic task value, that the property is guaranteed to hold for every task execution. We have proven the symbolic semantics to be sound and complete with respect to the original $\widehat{\text{TOP}}$ semantics.

To support workers using workflow software, we want to provide them with feedback specific to their situation. Decision support systems are proven to be effective in supporting users making choices in workflow systems. They come at a cost however. Significant investment is required in developing them, and once developed, these systems are very rigid. By answering the following question, we have attempted to alleviate these drawbacks by reducing the effort that is required in the development of such systems.

How can we calculate next-step hints from a workflow specification?

I have presented two different approaches to calculate next-step hints from a workflow specification.

Part I offers a framework to programmers to easily translate their workflow to make it tractable to generic solving. By describing the workflow in a special domain-specific language (DSL), we are able to run generic search algorithms on it. The DSL is constructed in such a way that it supports common workflow patterns. A goal is defined in terms of a predicate over the end state. Several search algorithms from the artificial intelligence domain are made available in the framework, to find traces that lead to the defined goal. From these traces, hint information can be generated.

To demonstrate that the framework is indeed able to deal with different kinds of problems, we have implemented several examples. These examples come from workflow systems, computer games and intelligent tutoring systems. These implementations show that our DLS is expressive enough to capture a wide variety of problems, and that the search algorithms included in the framework are adequate.

When comparing this solution to other workflow problem modelling languages, we find that our solution offers more flexibility, and allows programmers to better describe problems. Existing systems, like PDDL, SITPLAN, STRIPS and PLANNER often have big limitations, like finite space-state, finite set of conditions, or an absence of solving algorithms.

Part II takes a different approach. Instead of having a programmer translate a problem into a special DSL, we are able to calculate hints for any $\widehat{\text{TOP}}$ program, without alterations. To achieve this, we leverage symbolic execution. To calculate next-step hints for a $\widehat{\text{TOP}}$ program, first a goal over the resulting value of a program is formulated. The program is then symbolically executed. From all the symbolic runs, those that fulfil the goal condition are selected. By returning the input that each of those runs requires, we automatically calculate hints for any TOP program, without changing the program.

The advantage of this method is that absolutely no interference of the programmer is required. The program can remain unchanged, and any changes made to the program are directly taken into account when calculating hints, making this approach extremely flexible.

The automatic next-step hint system based on symbolic execution is shown to be sound and complete. This is done by first showing the symbolic execution semantics sound and complete, followed by a proof that the hints themselves are sound and complete.

This approach has several benefits when compared to traditional decision support systems (DSSs). There is no need to develop a model for each problem individually. Instead, the existing implementation can directly be used to calculate next-step hints. This makes our approach less costly, quicker to develop and more flexible.

Chapter 9

Future work

Doing science is like shining a spotlight into the dark. The more knowledge we uncover, the wider the beam becomes. Although the illuminated area becomes bigger, the area that we know we cannot see increases faster.

The same holds true for the research presented in this dissertation. The following sections list many suggestions for future work. They are divided over three main areas; end-user run-time feedback, task analysis, and TOP language development.

9.1 End-user run-time feedback

9.1.1 Unified hints framework

The most obvious next step would be to develop a framework that integrates the two next-step hint approaches. This would provide programmers with two different methods of providing hints to end users. By using RuleTrees, programmers can annotate their software. Doing this manually gives programmers control over the granularity of the hints. Using symbolic execution, hints can automatically be generated without programmer interference over every user step. This hybrid approach provides low- and high-level feedback to end users, making the run-time feedback even more tailored to their specific needs.

To give an example, imagine a workflow system that models a navy ship. The tasks model the movements and actions of workers on board. When a fire breaks out, calculating a hint from the RuleTree might result in next-step hints like "extinguish fire". Using Assistive $\widehat{\text{TOP}}$ on the other hand will result in a next-step hint on the level of individual steps, that in sequence could look something like this: "Move to room 1", "Pick up extinguisher", "Move to room 2", "Use extinguisher". This approach is similar to the annotations used in Haskell tutor Ask-Elle (Gerdes, Heeren, & Jeuring, 2012).

9.1.2 iTasks integration

Currently, iTasks is the most used TOP implementation. It would therefore be very interesting to bring the techniques described in this dissertation to iTasks. For the programmer assisted next-step hints, details on how this can be done are described in Section 2.6.2. This solution is quite ad-hoc however. The ruleTree structure could also be integrated in the iTasks language to allow programmers to integrate the rule-based problem description in the actual task specification.

The automatic hint generation system would be more challenging to integrate into iTasks. This method depends on symbolic execution, which has not been developed for Clean and iTasks. Since Clean support higher order constructs, it is non-trivial to develop a symbolic execution semantics for it. However, some research has been done to bring symbolic execution to higher-order languages, as described in Section 6.7.1.

9.1.3 Hint presentation

Current implementations of both the assisted and automatic systems are mere proof of concepts implementations. It is possible to calculate next-step hints, but there is currently no way to display hints in a user friendly manner. The information calculated by both systems potentially contains duplicate hints and redundant or irrelevant constraint information.

The same holds for the user defined goals, there is no user friendly way to set a goal. When implementing either or both of the hint frameworks into real-world applications, some research has to be done to determine how to display end-user hints and how to set goals.

9.1.4 Testing the effect of hints

The effectiveness of hints has been shown in other research, especially in the intelligent tutoring community Kulik and Fletcher (2016); Sharda, Barr, and McDonnell (1988). To validate the approaches proposed in this dissertation, it would be interesting to conduct empirical studies. This would allow us to determine the effectiveness of next-step hints. TOP research has been applied and studied in the field at the Royal Netherlands Navy and the Royal Netherlands Sea Rescue Institution. The existing systems implemented in TOP at these two institutes would be ideal testing ground for Assistive $\widehat{\text{TOP}}$.

9.1.5 Other kinds of feedback

In this dissertation, I focus mainly on providing next-step hints. Of course, there are many other possible forms of feedback.

In certain cases, it might be that a more general hint is more didactically effective. For example, when solving a math problem, it could be more useful to first tell a student what approach she could try, before actually suggesting a concrete step.

In interactive programs, it might be the case that certain steps are not available to a user. It would be useful to inform the user, why a step is unavailable. For example, it could be that she needs to wait on her colleague to perform some action.

A different angle would be to look at managers' information. It is possible to build a manager's overview with information on the progress of tasks in an ad-hoc manner, but we are also interested in developing a more generic way to offer managers feedback.

9.2 Task analysis

9.2.1 Analysis of TopHat programs

The development of a formal Task-oriented Programming semantics opens the door for many different formal method techniques to be applied this domain. We are interested in equality of two $\widehat{\text{TOP}}$ programs, for example to show that the monad laws hold for our step combinator. This requires a notion of equality, which in the presence of side effects most certainly needs some form of coalgebraic input-output conformance.

Another interesting question is investigating temporal properties. Currently, Symbolic $\widehat{\text{TOP}}$ only looks at the resulting task value. We would also like to be able to prove certain properties to hold during the entire execution. For example, thinking about the flight booking example, it should never be possible for two people to book the same seat. This property should not only hold at the end of the program, but also during execution.

Another form of reasoning about programs is static analysis. Klinik, Jansen, and Plasmeijer (2017) have developed a cost analysis for tasks that require resources in order to be executed. This analysis was developed for a simpler task language, and could be applied to the one developed here.

9.2.2 Verification of iTasks behaviour

In the development of $\widehat{\text{TOP}}$, we wanted to capture the essence of Task-oriented programming. By defining a formal semantics, we state clearly

and unambiguously how each combinator is expected to behave, and how evaluation reacts to incoming user events.

For the most used TOP system, iTasks, such a formal semantics is not available. The implementation in Clean is actually the language specification. It would be very interesting to use the formal $\widehat{\text{TOP}}$ semantics to verify the behaviour of iTasks. If we define a program in iTasks and in $\widehat{\text{TOP}}$, do they behave the same?

iTasks is designed with a different philosophy in mind than $\widehat{\text{TOP}}$. Instead of choosing the combinator set in such a way that they only perform one specific task, iTasks employs swiss-army-knife-like combinators that perform many actions at once.

The step combinator $>>*$ for example, is used to perform two tasks in sequence, but also allows users to choose from a list of alternatives, enables or disables those alternatives based on the previous task value, and allows the task to step to one of the alternatives based on the task value and without user interaction, all in one combinator.

In $\widehat{\text{TOP}}$, we have opted for combinators that only perform one task, and with the set of combinators, it should be possible to construct every iTask combinator behaviour. The combinators in $\widehat{\text{TOP}}$ are modelled after iTasks combinators, and several examples have been implemented to solidify our confidence, but proving complete coverage of iTasks behaviour is left as future work.

A different approach to verify iTasks behaviour would be to model the task in a separate system, and then during run-time check that the execution adheres to the model.

9.2.3 Workflow mining

In the workflow community, some research is done into workflow mining. The idea is that you take existing data, such as activity logs, and use big-data techniques to learn, or mine, a workflow. Usually, the form of the learned model is a workflow-net, or petri-net.

By choosing to learn a petri-net, it is only possible to mine limited workflows. For example, it is not possible to mine higher order workflows, since the visual petri-net representation does not support them. I would be very interested in mining a workflow and target the iTasks or $\widehat{\text{TOP}}$ form. My hypothesis is that the learned workflows are smaller, reusable, and more efficient than the workflows learned using petri-nets.

9.3 TOP language development

9.3.1 Visual TopHat

We would like to develop visualisations for $\widehat{\text{TOP}}$ language constructs. An assistive development environment integrating these visualisations and the presented textual language would aid domain experts to model workflows in a more accessible manner. A system that visualises iTask programs has been developed in the past (Stutterheim et al., 2014).

9.3.2 TopHat 2.0

While working with $\widehat{\text{TOP}}$, we discovered that certain design decisions were sub-optimal, and could be improved. More specifically, we intend to publish a new version of $\widehat{\text{TOP}}$ in the near future, containing the following improvements.

- Moving the references from the host language layer to the task layer. This reduces complexity.
- Removing the user step combinator from the language. This combinator is redundant, since it can be created by using the external choice combinator.
- Adding read-only editors. Many applications require a way to display information to the user, without allowing the user to modify the information. In current TOP implementations, this is achieved by using read-only editors.
- Adding a forever-combinator. Many tasks run indefinitely, $\widehat{\text{TOP}}$ does not currently model this behaviour. Think of a coffee machine that accepts a coin, dispenses coffee and then returns to its initial state.
- A new normalisation mechanism. Instead of comparing states and tasks to determine if a task has been normalised, the new semantics will keep track of which references are currently being watched, and whether or not they have changed.
- Dedicating a special editor to internal values, to make a distinction between values which are editable by the user, and other values that can be lifted into the task world, like functions and tasks themselves

Curriculum Vitae

16 august 1991	Born in Gorinchem, The Netherlands
2003 - 2011	VWO N&T N&G at GSG Leo Vroman in Gouda
2011 - 2013	Bachelor Computer Science at Utrecht University
2013 - 2015	Master Computing Science at Utrecht University
2014 - 2015	Internship at Albert Ludwigs University of Freiburg
2015 - 2020	PhD Candidate at Utrecht University
2017 - 2019	University Council member at Utrecht University
2020 - current	Post-Doc researcher at Open University

Appendices

Appendix A

TopHat type preservation proofs

A.1 Type preservation under evaluation

Proof: We prove Theorem 5.5.1 by induction on a derivation of $\Gamma, \Sigma \vdash e : \tau$:

Case T-CONSTBOOL, T-VAR, T-LOC, T-ABS, T-IF, T-APP, T-REF, T-DEREF, T-ASSIGN

Type preservation has been proven for these cases by Pierce (2002).

Case T-CONSTINT, T-CONSTSTRING, T-UNIT, T-LISTEMPTY, T-ENTER, T-XOR

Evaluation does not alter the expression and state, therefore these cases trivially hold.

Case T-FAIL

This case cannot happen, since there is no evaluation rule for fail.

Case T-FIRST, T-SECOND, T-LISTHEAD, T-LISTTAIL, T-EDIT, T-UPDATE, T-THEN, T-NEXT

These cases follow immediately from application of the induction hypothesis.

Case T-PAIR, T-CONS, T-OR, T-AND

These cases follow immediately by applying the induction hypothesis twice.

□

A.2 Type preservation under striding

Before we can prove type preservation under striding, we need to prove that the value function also preserves types.

A.2.1 Task value preserves types

Lemma A.2.1

For all expressions e and states σ such that $\Gamma, \Sigma \vdash e : \text{Task } \tau$ and $\Gamma, \Sigma \vdash \sigma$, if $\mathcal{V}(e, \sigma) = v$, then $v : \tau$.

Proof: We prove Lemma A.2.1 by induction over a derivation of $\Gamma, \Sigma \vdash e : \text{TASK } \tau$:

Case T-FAIL

$\mathcal{V}(\perp, \sigma) = \perp$. Since this case does not lead to a value, the lemma holds trivially.

Case T-EDIT

$\mathcal{V}(\Box v, \sigma) = v$. By T-Edit, if $\Gamma, \Sigma \vdash \Box v : \text{TASK } \beta$, then $\Gamma, \Sigma \vdash v : \beta$.

Case T-ENTER

$\mathcal{V}(\Box \beta, \sigma) = \perp$. Since this case does not lead to a value, the lemma holds trivially.

Case T-UPDATE

$\mathcal{V}(\blacksquare l, \sigma) = \sigma(l)$. Given that $\Gamma, \Sigma \vdash \blacksquare l : \text{TASK } \beta$, we know that $\Gamma, \Sigma \vdash l : \text{REF } \beta$ by the premise of T-UPDATE. By definition of a well typed state and given that $\Gamma, \Sigma \vdash \sigma$, we know that $\Gamma, \Sigma \vdash \sigma(l) : \Gamma(l)$. Then by T-LOC, we obtain that $\Gamma(l) = \beta$, and thus $\Gamma, \Sigma \vdash \sigma(l) : \beta$.

Case T-THEN

$\mathcal{V}(t_1 \blacktriangleright e_2, \sigma) = \perp$. Since this case does not lead to a value, the lemma holds trivially.

Case T-NEXT

$\mathcal{V}(t_2 \triangleright e_2, \sigma) = \perp$. Since this case does not lead to a value, the lemma holds trivially.

Case T-AND

Subcase $\mathcal{V}(t_1 \bowtie t_2, \sigma) = \langle v_1, v_2 \rangle$

Given that $\mathcal{V}(t_1, \sigma) = v_1 \wedge \mathcal{V}(t_2, \sigma) = v_2$. By T-AND we have that $\Gamma, \Sigma \vdash t_1 \bowtie t_2 : \text{TASK}(\tau_1 \times \tau_2)$ and $\Gamma, \Sigma \vdash t_1 : \tau_1$ and $\Gamma, \Sigma \vdash t_2 : \tau_2$. By the induction hypothesis, $\mathcal{V}(t_1, \sigma) = v_1$ and $\mathcal{V}(t_2, \sigma) = v_2$ preserve types, and thus $\Gamma, \Sigma \vdash v_1 : \tau_1$ and $\Gamma, \Sigma \vdash v_2 : \tau_2$. This gives us that $\Gamma, \Sigma \vdash \langle v_1, v_2 \rangle : \text{TASK}(\tau_1 \times \tau_2)$.

Subcase $\mathcal{V}(t_1 \bowtie t_2, \sigma) = \perp$

Given that $\neg(\mathcal{V}(t_1, \sigma) = v_1 \wedge \mathcal{V}(t_2, s) = v_2)$. Since this case does not lead to a value, the lemma holds trivially.

Case T-OR

Subcase $\mathcal{V}(t_1 \blacklozenge t_2, \sigma) = v_1$

Given that $\mathcal{V}(t_1, \sigma) = v_1$. By T-OR we have that $\Gamma, \Sigma \vdash t_1 \blacklozenge t_2 : \text{TASK } \tau$, and $\Gamma, \Sigma \vdash t_1 : \text{TASK } \tau$ and $\Gamma, \Sigma \vdash t_2 : \text{TASK } \tau$. By the induction hypothesis, we have that $\Gamma, \Sigma \vdash v_1 : \tau$.

Subcase $\mathcal{V}(t_1 \blacklozenge t_2, \sigma) = v_2$

Given that $\mathcal{V}(t_1, \sigma) = \perp \wedge \mathcal{V}(t_2, \sigma) = v_2$. By T-OR we have that $\Gamma, \Sigma \vdash t_1 \blacklozenge t_2 : \text{TASK } \tau$, and $\Gamma, \Sigma \vdash t_1 : \text{TASK } \tau$ and $\Gamma, \Sigma \vdash t_2 : \text{TASK } \tau$. By the induction hypothesis, we have that $\Gamma, \Sigma \vdash v_2 : \tau$.

Subcase $\mathcal{V}(t_1 \blacklozenge t_2, \sigma) = \perp$

Given that $\mathcal{V}(t_1, \sigma) = \perp \wedge \mathcal{V}(t_2, \sigma) = \perp$. Since this case does not lead to a value, the lemma holds trivially.

Case T-XOR

$\mathcal{V}(t_1 \diamond t_2, \sigma) = \perp$. Since this case does not lead to a value, the lemma holds trivially. □

A.2.2 Striding preserves types

Lemma A.2.2

For all expressions e and states σ such that $\Gamma, \Sigma \vdash e : \text{TASK } \tau$ and $\Gamma, \Sigma \vdash \sigma$, if $e, \sigma \mapsto e', \sigma'$, then $\Gamma, \Sigma \vdash e' : \text{TASK } \tau$ and $\Gamma, \Sigma \vdash \sigma'$.

Proof: We prove Lemma A.2.2 by induction on a derivation of $\Gamma, \Sigma \vdash e : \text{TASK } \tau$:

Case T-FAIL, T-EDIT, T-ENTER, T-UPDATE, T-XOR

Since these cases do not alter the expression, the theorem trivially holds.

Case T-AND

Given that $\Gamma, \Sigma \vdash t_1 \bowtie t_2 : \text{TASK}(\tau_1 \times \tau_2)$, by T-AND we have $\Gamma, \Sigma \vdash t_1 : \text{TASK } \tau_1$ and $\Gamma, \Sigma \vdash t_2 : \text{TASK } \tau_2$. By the induction hypothesis we have $\Gamma, \Sigma \vdash t'_1 : \text{TASK } \tau_1$ and $\Gamma, \Sigma \vdash \sigma'$, and $\Gamma, \Sigma \vdash t'_2 : \text{TASK } \tau_2$ and $\Gamma, \Sigma \vdash \sigma''$. This gives us that $\Gamma, \Sigma \vdash t'_1 \bowtie t'_2 : \text{TASK}(\tau_1 \times \tau_2)$ by T-AND.

Case T-NEXT

Given that $\Gamma, \Sigma \vdash t_1 \triangleright e_2 : \text{TASK } \tau$, T-THEN gives us that $\Gamma, \Sigma \vdash t_1 : \text{TASK } \tau_1$ and $\Gamma, \Sigma \vdash e_2 : \tau_1 \rightarrow \text{TASK } \tau$. By the induction hypothesis, we know that $t_1, \sigma \mapsto t'_1, \sigma'$ preserves types and thus $\Gamma, \Sigma \vdash t'_1 : \text{TASK } \tau_1$ and $\Gamma, \Sigma \vdash \sigma'$. Therefore $\Gamma, \Sigma \vdash t'_1 \triangleright e_2 : \text{TASK } \tau$.

Case T-OR**Subcase S-ORLEFT**

Given that $\Gamma, \Sigma \vdash t_1 \blacklozenge t_2 : \text{TASK } \tau$, by T-OR we have $\Gamma, \Sigma \vdash t_1 : \text{TASK } \tau$. By the induction hypothesis, we know that $t_1, \sigma \mapsto t'_1, \sigma'$ preserves types and thus $\Gamma, \Sigma \vdash t'_1 : \text{TASK } \tau$ and $\Gamma, \Sigma \vdash \sigma'$.

Subcase S-ORRIGHT

Given that $\Gamma, \Sigma \vdash t_1 \blacklozenge t_2 : \text{TASK } \tau$, by T-OR we have $\Gamma, \Sigma \vdash t_2 : \text{TASK } \tau$. By the induction hypothesis, we know that $t_2, \sigma \mapsto t'_2, \sigma'$ preserves types and thus $\Gamma, \Sigma \vdash t'_2 : \text{TASK } \tau$ and $\Gamma, \Sigma \vdash \sigma'$.

Subcase S-ORNONE

Given that $\Gamma, \Sigma \vdash t_1 \blacklozenge t_2 : \text{TASK } \tau$, by T-OR we have $\Gamma, \Sigma \vdash t_1 : \text{TASK } \tau$ and $\Gamma, \Sigma \vdash t_2 : \text{TASK } \tau$. By the induction hypothesis, we know that $t_1, \sigma \mapsto t'_1, \sigma'$ and $t_2, \sigma' \mapsto t'_2, \sigma''$ preserve types, and thus $\Gamma, \Sigma \vdash t'_1 \blacklozenge t'_2 : \text{TASK } \tau$ and $\Gamma, \Sigma \vdash \sigma''$.

Case T-THEN**Subcase S-THENSTAY**

Given that $\Gamma, \Sigma \vdash t_1 \blacktriangleright e_2 : \text{TASK } \tau$, by T-THEN we have $\Gamma, \Sigma \vdash t_1 : \text{TASK } \tau_1$ and $\Gamma, \Sigma \vdash e_2 : \tau_1 \rightarrow \text{TASK } \tau$. By the induction hypothesis, we know that $t_1, \sigma \mapsto t'_1, \sigma'$ preserves types, and thus $\Gamma, \Sigma \vdash t'_1 \blacktriangleright e_2 : \text{TASK } \tau$ and $\Gamma, \Sigma \vdash \sigma'$.

Subcase S-THENFAIL

Given that $\Gamma, \Sigma \vdash t_1 \blacktriangleright e_2 : \text{TASK } \tau$, by T-THEN we have $\Gamma, \Sigma \vdash t_1 : \text{TASK } \tau_1$ and $e_2 : \tau_1 \rightarrow \text{TASK } \tau$. By the induction hypothesis, we know that $t_1, \sigma \mapsto t'_1, \sigma'$ preserves types, and thus $\Gamma, \Sigma \vdash t'_1 \blacktriangleright e_2 : \text{TASK } \tau$ and $\Gamma, \Sigma \vdash \sigma'$.

Subcase S-THENCONT

Given that $\Gamma, \Sigma \vdash t_1 \blacktriangleright e_2 : \text{TASK } \tau$, by T-THEN we have $\Gamma, \Sigma \vdash t_1 : \text{TASK } \tau_1$ and $\Gamma, \Sigma \vdash e_2 : \tau_1 \rightarrow \text{TASK } \tau$. By the induction hypothesis, we know that $t_1, \sigma \mapsto t'_1, \sigma'$ preserves types. By Lemma A.2.1, we know that $\mathcal{V}(t'_1, \sigma') = v_1$ preserves types. By Theorem 5.5.1 we know that $e_2 v_1, \sigma' \downarrow t_2, \sigma''$ preserves types. Therefore $\Gamma, \Sigma \vdash t_2 : \text{TASK } \tau$ and $\Gamma, \Sigma \vdash \sigma''$.

□

A.3 Proof of type preservation under normalisation

Proof: We prove Theorem 5.5.2 by induction on the derivation of e :

$$\begin{array}{c}
 \text{N-DONE} \\
 e, \sigma \downarrow t, \sigma' \\
 \text{Case } t, \sigma' \mapsto t', \sigma'' \\
 \frac{\sigma' = \sigma'' \wedge t = t'}{e, \sigma \Downarrow t, \sigma'}
 \end{array}$$

Given that $\Gamma, \Sigma \vdash e : \text{TASK } \tau$ and $\Gamma, \Sigma \vdash \sigma$, we know that $\Gamma, \Sigma \vdash t : \text{TASK } \tau$ and $\Gamma, \Sigma \vdash \sigma'$ by Theorem 5.5.1.

$$\begin{array}{c}
 \text{N-REPEAT} \\
 e, \sigma \downarrow t, \sigma' \\
 \text{Case } t, \sigma' \mapsto t', \sigma'' \\
 \sigma' \neq \sigma'' \vee t \neq t' \\
 \frac{t', \sigma'' \Downarrow t'', \sigma'''}{e, \sigma \Downarrow t'', \sigma'''}
 \end{array}$$

Given that $\Gamma, \Sigma \vdash e : \text{TASK } \tau$ and $\Gamma, \Sigma \vdash \sigma$, we know that $\Gamma, \Sigma \vdash t : \text{TASK } \tau$ and $\Gamma, \Sigma \vdash \sigma'$ by Theorem 5.5.1. Then by Lemma A.2.2, we have $\Gamma, \Sigma \vdash t' : \text{TASK } \tau$ and $\Gamma, \Sigma \vdash \sigma''$. Then by the induction hypothesis, we finally obtain that $\Gamma, \Sigma \vdash t'' : \text{TASK } \tau$ and $\Gamma, \Sigma \vdash \sigma'''$.

□

A.4 Proof of type preservation under handling

We require the following Lemma for this proof.

Lemma A.4.1

Given that $\Gamma, \Sigma \vdash \sigma, \Sigma(l) = \tau$ and $\Gamma, \Sigma \vdash v : \tau$, it holds that $\Sigma \vdash \sigma[l \mapsto v]$

This lemma follows immediately from the definition of a well typed state.

Proof: We prove Theorem 5.5.3 by induction on a derivation of $\Gamma, \Sigma \vdash e : \text{TASK } \tau$:

Case T-FAIL

This case cannot happen.

Case T-EDIT

Given that $\Gamma, \Sigma \vdash \Box v : \text{TASK } \beta$ and $\Gamma, \Sigma \vdash \sigma$, the H-CHANGE rule additionally gives us that $v, v' : \beta$. Therefore by T-EDIT we have that $\Gamma, \Sigma \vdash \Box v' : \text{TASK } \beta$.

Case T-ENTER

Given that $\Gamma, \Sigma \vdash \boxtimes \beta : \text{Task } \beta$ and $\Gamma, \Sigma \vdash \sigma$, the H-FILL rule additionally gives us that $v : \beta$. Then by T-ENTER we have $\Gamma, \Sigma \vdash \square v : \text{Task } \beta$.

Case T-UPDATE

$\Gamma, \Sigma \vdash \blacksquare l : \text{Task } \beta$ and $\Gamma, \Sigma \vdash \sigma$. This gives us that $\Sigma(l) = \beta$, and we additionally obtain $\sigma(l), v : \beta$ by H-UPDATE. By application of Lemma A.4.1 this case holds.

Case T-XOR**Subcase H-PICKLEFT**

Given that $\Gamma, \Sigma \vdash t_1 \diamond t_2 : \text{Task } \tau$ and $\Gamma, \Sigma \vdash \sigma$, then by Theorem 5.5.2, we have $\Gamma, \Sigma \vdash t_1 : \text{Task } \tau$ and $\Gamma, \Sigma \vdash \sigma'$.

Subcase H-PICKRIGHT

Given that $\Gamma, \Sigma \vdash t_1 \diamond t_2 : \text{Task } \tau$ and $\Gamma, \Sigma \vdash \sigma$, then by Theorem 5.5.2, we have $\Gamma, \Sigma \vdash t_2 : \text{Task } \tau$ and $\Gamma, \Sigma \vdash \sigma'$.

Case T-NEXT**Subcase H-NEXT**

Given that $\Gamma, \Sigma \vdash t_1 \triangleright e_2 : \text{Task } \tau$ and $\Gamma, \Sigma \vdash \sigma$, then by Lemma A.2.1 and Theorem 5.5.2, we have $\Gamma, \Sigma \vdash t_2 : \text{Task } \tau$ and $\Gamma, \Sigma \vdash \sigma'$.

Subcase H-PASSNEXT

Given that $\Gamma, \Sigma \vdash t_1 \triangleright e_2 : \text{Task } \tau$ and $\Gamma, \Sigma \vdash \sigma$, T-NEXT gives us that $\Gamma, \Sigma \vdash t_1 : \text{Task } \tau_1$ and $\Gamma, \Sigma \vdash e_2 : \tau_1 \rightarrow \text{Task } \tau$. By the induction hypothesis, we know that $t_1, \sigma \xrightarrow{i} t'_1, \sigma'$ also preserves types and thus $\Gamma, \Sigma \vdash t'_1 : \text{Task } \tau_1$ and $\Gamma, \Sigma \vdash \sigma'$. By T-NEXT we now obtain that $\Gamma, \Sigma \vdash t'_1 \triangleright e_2 : \text{Task } \tau$.

Case T-THEN**H-PASSTHEN**

Given that $\Gamma, \Sigma \vdash t_1 \blacktriangleright e_2 : \text{Task } \tau$ and $\Gamma, \Sigma \vdash \sigma$, T-THEN gives us that $\Gamma, \Sigma \vdash t_1 : \text{Task } \tau_1$ and $\Gamma, \Sigma \vdash e_2 : \tau_1 \rightarrow \text{Task } \tau$. By the induction hypothesis, we know that $t_1, \sigma \xrightarrow{i} t'_1, \sigma'$ also preserves types and thus $\Gamma, \Sigma \vdash t'_1 : \text{Task } \tau_1$ and $\Gamma, \Sigma \vdash \sigma'$. By T-THEN we now obtain that $\Gamma, \Sigma \vdash t'_1 \blacktriangleright e_2 : \text{Task } \tau$.

Case T-AND**Subcase H-FIRSTAND**

Given that $\Gamma, \Sigma \vdash t_1 \bowtie t_2 : \text{TASK}(\tau_1 \times \tau_2)$ and $\Gamma, \Sigma \vdash \sigma$, T-AND gives us that $\Gamma, \Sigma \vdash t_1 : \text{TASK } \tau_1$ and $\Gamma, \Sigma \vdash t_2 : \text{TASK } \tau_2$. By the induction hypothesis, we know that $t_1, \sigma \xrightarrow{i} t'_1, \sigma'$ also preserves types and thus $\Gamma, \Sigma \vdash t'_1 : \text{TASK } \tau_1$ and $\Gamma, \Sigma \vdash \sigma'$. Therefore by T-AND we obtain $\Gamma, \Sigma \vdash t'_1 \bowtie t_2 : \text{TASK}(\tau_1 \times \tau_2)$.

Subcase H-SECONDAND

Given that $\Gamma, \Sigma \vdash t_1 \bowtie t_2 : \text{TASK}(\tau_1 \times \tau_2)$ and $\Gamma, \Sigma \vdash \sigma$, T-AND gives us that $\Gamma, \Sigma \vdash t_1 : \text{TASK } \tau_1$ and $\Gamma, \Sigma \vdash t_2 : \text{TASK } \tau_2$. By the induction hypothesis, we know that $t_2, \sigma \xrightarrow{i} t'_2, \sigma'$ also preserves types and thus $\Gamma, \Sigma \vdash t'_2 : \text{TASK } \tau_2$ and $\Gamma, \Sigma \vdash \sigma'$. Therefore by T-AND we obtain $\Gamma, \Sigma \vdash t_1 \bowtie t'_2 : \text{TASK}(\tau_1 \times \tau_2)$.

Case T-OR

Subcase H-FIRSTOR

Given that $\Gamma, \Sigma \vdash t_1 \blacklozenge t_2 : \text{TASK } \tau$ and $\Gamma, \Sigma \vdash \sigma$, T-OR gives us that $\Gamma, \Sigma \vdash t_1 : \text{TASK } \tau$ and $\Gamma, \Sigma \vdash t_2 : \text{TASK } \tau$. By the induction hypothesis we know that $t_1, \sigma \xrightarrow{i} t'_1, \sigma'$ also preserves types, and therefore $\Gamma, \Sigma \vdash t'_1 : \text{TASK } \tau$ and $\Gamma, \Sigma \vdash \sigma'$. By T-OR we now obtain $\Gamma, \Sigma \vdash t'_1 \blacklozenge t_2 : \text{TASK } \tau$.

Subcase H-SECONDOR

Given that $\Gamma, \Sigma \vdash t_1 \blacklozenge t_2 : \text{TASK } \tau$ and $\Gamma, \Sigma \vdash \sigma$, T-OR gives us that $\Gamma, \Sigma \vdash t_1 : \text{TASK } \tau$ and $\Gamma, \Sigma \vdash t_2 : \text{TASK } \tau$. By the induction hypothesis we know that $t_2, \sigma \xrightarrow{i} t'_2, \sigma'$ also preserves types, and therefore $\Gamma, \Sigma \vdash t'_2 : \text{TASK } \tau$ and $\Gamma, \Sigma \vdash \sigma'$. By T-OR we now obtain $\Gamma, \Sigma \vdash t_1 \blacklozenge t'_2 : \text{TASK } \tau$.

□

Appendix B

TopHat progress proof

Proof: We prove Theorem 5.5.5 by induction on a derivation of $\Gamma, \Sigma \vdash e : \text{TASK } \tau$:

Case T-FAIL

$\mathcal{F}(\perp, \sigma) = \text{True}$, and there is no handling rule that applies to fail.

Case T-EDIT

$\mathcal{F}(\Box v, \sigma) = \text{False}$, and there exists an input i , namely any value v' of type β , that can be handled.

Case T-ENTER

$\mathcal{F}(\boxtimes \beta, \sigma) = \text{False}$, and there exists an input i , namely any value v of type β , that can be handled.

Case T-UPDATE

$\mathcal{F}(\blacksquare l, \sigma) = \text{False}$, and there exists an input i , namely any value v of type β , that can be handled.

Case T-THEN

$\mathcal{F}(t_1 \blacktriangleright e_2, \sigma) = \mathcal{F}(t_1, \sigma)$. If there exists an i for t_1 , then this i also applies to $t_1 \blacktriangleright e_2$. This case therefore holds by the induction hypothesis.

Case T-NEXT

$\mathcal{F}(t_1 \triangleright e_2, \sigma) = \mathcal{F}(t_1, \sigma)$. If there exists an i for t_1 , then this i also applies to $t_1 \triangleright e_2$. This case therefore holds by the induction hypothesis.

Case T-XOR

We normalise the two expressions first, $e_1, \sigma \Downarrow t_1, \sigma', e_2, \sigma \Downarrow t_2, \sigma'$ and we can then be in two situations. One, we can have that $\mathcal{F}(t_1, \sigma')$ and $\mathcal{F}(t_2, \sigma')$ are both true. If that is so, then by definition, we have both $\mathcal{F}(e_1 \diamond e_2, \sigma)$ and no rule of the handling semantics applies, and therefore there exists no input for this case.

Or we are in the situation where one or both of the sub-expressions does not fail. In that case, we know that $\mathcal{F}(e_1 \diamond e_2, \sigma)$ does not hold, and that at least one of the handling rules applies. Therefore, there must be an input i , namely L, R or both.

Case T-AND

We can again find ourselves in one of two situations. In the first case, both sub-expressions fail, $\mathcal{F}(t_1, \sigma)$ and $\mathcal{F}(t_2, \sigma)$. In that case, we know that $\mathcal{F}(t_1 \bowtie t_2, \sigma)$ also fails by definition. By the induction hypothesis, we know that for both t_1 and t_2 there is no input that can be handled.

Since the only two rules for $\bowtie$ that handle input just pass this input on to one of the two expressions, we know that indeed no i applies.

In the case that one or both sub-expressions do not fail, then by definition $t_1 \bowtie t_2$ is not failing under σ . Again by the induction hypothesis, we know that for one or both of the expressions, there exists an i that can be handled. Then by H-FirstAnd and H-SecondAnd, we know that we can pass this i , by prefixing it with either F or S.

Case T-OR

We can again find ourselves in one of two situations. In the first case, both sub-expressions fail, $\mathcal{F}(t_1, \sigma)$ and $\mathcal{F}(t_2, \sigma)$. In that case, we know that $\mathcal{F}(t_1 \blacklozenge t_2, \sigma)$ also fails by definition. By the induction hypothesis, we know that for both t_1 and t_2 there is no input that can be handled. Since the only two rules for $\blacklozenge$ that handle input just pass this input on to one of the two expressions, we know that indeed no i applies.

In the case that one or both sub-expressions do not fail, then by definition $t_1 \blacklozenge t_2$ is not failing under σ . Again by the induction hypothesis, we know that for one or both of the expressions, there exists an i that can be handled. Then by H-FirstOr and H-SecondOr, we know that we can pass this i , by prefixing it with either F or S.

□

Appendix C

Proof of correctness of Inputs function

Proof: We prove Theorem 5.5.6 by induction over a derivation of $\Gamma, \Sigma \vdash e : \text{TASK } \tau$:

Case T-EDIT

The rule H-CHANGE applies, which gives us that $\Box v, \sigma \xrightarrow{v'} \Box v', \sigma$ with $v, v' : \beta$. We have by definition of the inputs function that $\mathcal{I}(\Box v : \text{TASK } \beta, \sigma) = \{v' : \beta\}$, which includes $v' : \beta$.

Case T-ENTER

The rule H-FILL applies, which gives us that $\boxtimes \beta, \sigma \xrightarrow{v} \Box v, \sigma$ with $v : \beta$. We have by definition of the inputs function that $\mathcal{I}(\boxtimes \beta, \sigma) = \{v : \beta\}$, which includes $v : \beta$.

Case T-UPDATE

The rule H-UPDATE applies, which gives us that $\blacksquare l, \sigma \xrightarrow{v} \blacksquare l, \sigma[l \mapsto v]$ with $\sigma(l), v : \beta$. We have by definition of the inputs function that $\mathcal{I}(\blacksquare l : \text{TASK } \beta, \sigma) = \{v : \beta\}$, which includes $v : \beta$.

Case T-FAIL

No handling rule applies in this case, and we have by definition of the inputs function that $\mathcal{I}(\text{fail}, \sigma) = \emptyset$.

Case T-XOR

Subcase $i = L$

The rule H-PICKLEFT applies and gives us $e_1 \diamond e_2, \sigma \xrightarrow{L} t_1, \sigma'$. We have by definition of the inputs function that $\mathcal{I}(e_1 \diamond e_2, \sigma) = \{L \mid e_1, \sigma \Downarrow t_1, \sigma' \wedge \neg \mathcal{F}(t_1, \sigma')\} \cup \{R \mid e_2, \sigma \Downarrow t_2, \sigma' \wedge \neg \mathcal{F}(t_2, \sigma')\}$. From the premise of H-PICKLEFT we can conclude that the conditions $e_1, \sigma \Downarrow t_1, \sigma' \wedge \neg \mathcal{F}(t_1, \sigma')$ hold, so we obtain L.

Subcase $i = R$

The rule H-PICKRIGHT applies and gives us $e_1 \diamond e_2, \sigma \xrightarrow{R} t_2, \sigma'$. We have by definition of the inputs function that $\mathcal{I}(e_1 \diamond e_2, \sigma) = \{L \mid e_1, \sigma \Downarrow t_1, \sigma' \wedge \neg \mathcal{F}(t_1, \sigma')\} \cup \{R \mid e_2, \sigma \Downarrow t_2, \sigma' \wedge \neg \mathcal{F}(t_2, \sigma')\}$. From the premise of H-PICKRIGHT we can conclude that the conditions $e_2, \sigma \Downarrow t_2, \sigma' \wedge \neg \mathcal{F}(t_2, \sigma')$ hold, so we obtain R.

Case T-NEXT

Subcase $i = C$

The rule H-NEXT applies, which gives us that $t_1 \triangleright e_2, \sigma \xrightarrow{C} t_2, \sigma'$. We have by definition of the inputs function that $\mathcal{I}(t_1 \triangleright e_2, \sigma) =$

$\mathcal{I}(t_1, \sigma) \cup \{C \mid \mathcal{V}(t_1, \sigma) = v_1 \wedge e_2 \ v_1, \sigma \Downarrow t_2, \sigma' \wedge \neg \mathcal{F}(t_2, \sigma')\}$.
 From the premise of H-NEXT we can conclude that the conditions $\mathcal{V}(t_1, s) = v_1 \wedge \neg \mathcal{F}(e_2 v_1, s \mapsto)$ hold, and therefore C is contained in the inputs.

Subcase $i \neq C$

The rule H-PASSNEXT applies and gives us $t_1 \triangleright e_2, \sigma \xrightarrow{i} t'_1 \triangleright e_2, \sigma'$.
 We have by definition of the inputs function that $\mathcal{I}(t_1 \triangleright e_2, s) = \mathcal{I}(t_1, \sigma) \cup \{C \mid \mathcal{V}(t_1, \sigma) = v_1 \wedge e_2 \ v_1, \sigma \Downarrow t_2, \sigma' \wedge \neg \mathcal{F}(t_2, \sigma')\}$.
 By the induction hypothesis, we have that $i \in \mathcal{I}(t_1, s)$, and by definition of $\mathcal{I}$, i is therefore also included in this case.

Case T-THEN

The rule H-PASSTHEN applies and gives us that $t_1 \blacktriangleright e_2, \sigma \xrightarrow{i} t'_1 \blacktriangleright e_2, \sigma'$.
 We have by definition of the inputs function that $\mathcal{I}(t_1 \blacktriangleright e_2, s) = \mathcal{I}(t_1, \sigma)$.
 By the induction hypothesis, we have that $i \in \mathcal{I}(t_1, s)$, and by definition of $\mathcal{I}$, i is therefore also included in this case.

Case T-AND

Subcase $i = Fi$

The rule H-FIRSTAND applies and gives us $t_1 \bowtie t_2, \sigma \xrightarrow{Fi} t'_1 \bowtie t_2, \sigma'$.
 We have by definition of the inputs function that $\mathcal{I}(t_1 \bowtie t_2, s) = \{Fi \mid i \in \mathcal{I}(t_1, s)\} \cup \{Si \mid i \in \mathcal{I}(t_2, \sigma)\}$. By the induction hypothesis, we have that $i \in \mathcal{I}(t_1, \sigma)$, and by definition of $\mathcal{I}$, Fi is therefore also included in this case.

Subcase $i = Si$

The rule H-SECONDAND applies and we therefore obtain that $t_1 \bowtie t_2, \sigma \xrightarrow{Si} t_1 \bowtie t'_2, \sigma'$. We have by definition of the inputs function that $\mathcal{I}(t_1 \bowtie t_2) = \{Fi \mid i \in \mathcal{I}(t_1, \sigma)\} \cup \{Si \mid i \in \mathcal{I}(t_2, \sigma)\}$.
 By the induction hypothesis, we have that $i \in \mathcal{I}(t_2, \sigma)$, and by definition of $\mathcal{I}$, Si is therefore also included in this case.

Case T-OR

Subcase $i = Fi$

The rule H-FIRSTOR applies and gives us $t_1 \blacklozenge t_2, \sigma \xrightarrow{Fi} t'_1 \blacklozenge t_2, \sigma'$. We have by definition of the inputs function that $\mathcal{I}(t_1 \blacklozenge t_2, \sigma) = \{Fi \mid i \in \mathcal{I}(t_1, \sigma)\} \cup \{Si \mid i \in \mathcal{I}(t_2, \sigma)\}$. By the induction hypothesis, we have that $i \in \mathcal{I}(t_1, \sigma)$, and by definition of $\mathcal{I}$, Fi is therefore also included in this case.

Subcase $i = Si$

The rule H-FIRSTOR applies and gives us $t_1 \blacklozenge t_2, \sigma \xrightarrow{Si} t_1 \blacklozenge t'_2, \sigma'$. We have by definition of the inputs function that $\mathcal{I}(t_1 \blacklozenge t_2, \sigma) = \{F\ i \mid i \in \mathcal{I}(t_1, \sigma)\} \cup \{S\ i \mid i \in \mathcal{I}(t_2, \sigma)\}$. By the induction hypothesis, we have that $i \in \mathcal{I}(t_2, \sigma)$, and by definition of $\mathcal{I}$, Si is therefore also included in this case.

□

Appendix D

Symbolic TopHat soundness and completeness

D.1 Soundness proofs

D.1.1 Proof of soundness of symbolic evaluation semantics

Proof: We prove Lemma 6.5.5 by induction over the derivation of the symbolic evaluation $e, \sigma \Downarrow \tilde{e}, \tilde{\sigma}, \tilde{\phi}$.

Case SE-VALUE

Since this case does not generate constraints, any M will do. Since neither the state, nor the expression is altered by the evaluation rule E-VALUE, this case holds trivially.

Case SE-FAIL

Since this case does not generate constraints, any M will do. Since neither the state, nor the expression $\perp$ is altered by the evaluation rule E-FAIL, this case holds trivially.

Case SE-PAIR

For all mappings M such that $M(\phi_1 \wedge \phi_2)$, we need to demonstrate that $\langle e_1, e_2 \rangle, \sigma \Downarrow \langle v_1, v_2 \rangle, \sigma''$ with $M\langle \tilde{v}_1, \tilde{v}_2 \rangle \equiv \langle v_1, v_2 \rangle$ and $M\tilde{\sigma}'' \equiv \sigma''$.

From the induction hypothesis, we obtain the following.

$\forall M_1. \tilde{e}_1, \tilde{\sigma} \Downarrow \tilde{v}_1, \tilde{\sigma}', \phi_1 \wedge M_1 \phi_1 \Rightarrow e_1, \sigma \Downarrow v_1, \sigma' \wedge M_1 \tilde{v}_1 \equiv v_1 \wedge M_1 \tilde{\sigma}' \equiv \sigma'$ and

$\forall M_2. M_2 \phi_2 \Rightarrow e_2, \sigma' \Downarrow v_2, \sigma'' \wedge M_2 \tilde{v}_2 \equiv v_2 \wedge M_2 \tilde{\sigma}'' \equiv \sigma''$.

Note that we have omitted from the second application of the induction hypothesis, the requirement that the symbolic step exists. The fact that this step exists is obtained from SE-PAIR and omitted to increase readability of this and any following proofs.

Since M satisfies both ϕ_1 and ϕ_2 , we obtain from E-PAIR and the induction steps above that $\langle e_1, e_2 \rangle, \sigma \Downarrow \langle v_1, v_2 \rangle, \sigma''$, $M\langle \tilde{v}_1, \tilde{v}_2 \rangle \equiv \langle v_1, v_2 \rangle$ and $M\tilde{\sigma}'' \equiv \sigma''$.

Case SE-FIRST

For all mappings M such that $M\phi$, we need to show that $\text{fst } e, \sigma \Downarrow v_1, \sigma'$ with $M\tilde{v}_1 \equiv v_1$ and $M\tilde{\sigma}' \equiv \sigma'$.

From the induction hypothesis, we obtain the following.

$\forall M_1. M_1 \phi \Rightarrow e, \sigma \Downarrow \langle v_1, v_2 \rangle, \sigma' \wedge M_1 \langle \tilde{v}_1, \tilde{v}_2 \rangle \equiv \langle v_1, v_2 \rangle \wedge M_1 \tilde{\sigma}' \equiv \sigma'$

Since M satisfies ϕ , we obtain from E-FIRST and the induction step above that $\text{fst } e, \sigma \Downarrow v_1, \sigma'$ with $M\tilde{v}_1 \equiv v_1$ and $M\tilde{\sigma}' \equiv \sigma'$.

Case SE-SECOND

For all mappings M such that $M\phi$, we need to show that $\text{snd } e, \sigma \Downarrow v_2, \sigma'$ with $M\tilde{v}_2 \equiv v_2$ and $M\tilde{\sigma}' \equiv \sigma'$.

From the induction hypothesis, we obtain the following.

$$\forall M_1. M_1\phi \Rightarrow e, \sigma \downarrow \langle v_1, v_2 \rangle, \sigma' \wedge M_1 \langle \tilde{v}_1, \tilde{v}_2 \rangle \equiv \langle v_1, v_2 \rangle \wedge M_1 \tilde{\sigma}' \equiv \sigma'$$

Since M satisfies ϕ , we obtain from E-SECOND and the induction step above that $\text{snd } e, \sigma \downarrow v_2, \sigma'$ with $M\tilde{v}_2 \equiv v_2$ and $M\tilde{\sigma}' \equiv \sigma'$.

Case SE-CONS

For all mappings M such that $M\phi$, we need to demonstrate that $e_1 :: e_2, \sigma \downarrow v_1 :: v_2, \sigma''$ with $M\tilde{v}_1 :: \tilde{v}_2 \equiv v_1 :: v_2$ and $M\tilde{\sigma}'' \equiv \sigma''$.

From the induction hypothesis, we obtain the following.

$$\forall M_1. M_1\phi_1 \Rightarrow e_1, \sigma \downarrow v_1, \sigma' \wedge M_1 \tilde{v}_1 \equiv v_1 \wedge M_1 \tilde{\sigma}' \equiv \sigma' \text{ and}$$

$$\forall M_2. M_2\phi_2 \Rightarrow e_2, \sigma' \downarrow v_2, \sigma'' \wedge M_2 \tilde{v}_2 \equiv v_2 \wedge M_2 \tilde{\sigma}'' \equiv \sigma''$$

Since M satisfies both ϕ_1 and ϕ_2 , we obtain from E-CONS and the induction steps above that $e_1 :: e_2, \sigma \downarrow v_1 :: v_2, \sigma''$ with $M(\tilde{v}_1 :: \tilde{v}_2) \equiv v_1 :: v_2$ and $M\tilde{\sigma}'' \equiv \sigma''$.

Case SE-HEAD

For all mappings M such that $M\phi$, we need to show that $\text{head } e, \sigma \downarrow v_1, \sigma'$ with $M\tilde{v}_1 \equiv v_1$ and $M\tilde{\sigma}' \equiv \sigma'$.

From the induction hypothesis, we obtain the following.

$$\forall M_1. M_1\phi \Rightarrow e, \sigma \downarrow v_1 :: v_2, \sigma' \wedge M_1(\tilde{v}_1 :: \tilde{v}_2) \equiv v_1 :: v_2 \wedge M_1 \tilde{\sigma}' \equiv \sigma'$$

Since M satisfies ϕ , we obtain from E-HEAD and the induction step above that $\text{head } e, \sigma \downarrow v_1, \sigma'$ with $M\tilde{v}_1 \equiv v_1$ and $M\tilde{\sigma}' \equiv \sigma'$.

Case SE-TAIL

For all mappings M such that $M\phi$, we need to show that $\text{tail } e, \sigma \downarrow v_2, \sigma'$ with $M\tilde{v}_2 \equiv v_2$ and $M\tilde{\sigma}' \equiv \sigma'$.

From the induction hypothesis, we obtain the following.

$$\forall M_1. M_1\phi \Rightarrow e, \sigma \downarrow v_1 :: v_2, \sigma' \wedge M_1(\tilde{v}_1 :: \tilde{v}_2) \equiv v_1 :: v_2 \wedge M_1 \tilde{\sigma}' \equiv \sigma'$$

Since M satisfies ϕ , we obtain from E-TAIL and the induction step above that $\text{tail } e, \sigma \downarrow v_2, \sigma'$ with $M\tilde{v}_2 \equiv v_2$ and $M\tilde{\sigma}' \equiv \sigma'$.

Case SE-APP

For all mappings M such that $M(\phi_1 \wedge \phi_2 \wedge \phi_3)$, we need to demonstrate that $e_1 e_2, \sigma \downarrow v_1, \sigma'''$ with $M\tilde{v}_1 \equiv v_1$ and $M\tilde{\sigma}''' \equiv \sigma'''$.

From the induction hypothesis, we obtain the following.

$$\forall M_1. M_1\phi_1 \Rightarrow e_1, \sigma \downarrow \lambda x : \tau. e'_1, \sigma' \wedge M_1 \lambda x : \tau. \tilde{e}'_1 \equiv \lambda x : \tau. e'_1 \wedge M_1 \tilde{\sigma}' \equiv \sigma' \text{ and}$$

$$\forall M_2. M_2\phi_2 \Rightarrow e_2, \sigma' \downarrow v_2, \sigma'' \wedge M_2 \tilde{v}_2 \equiv v_2 \wedge M_2 \tilde{\sigma}'' \equiv \sigma''$$

$$\text{and } \forall M_3. M_3\phi_3 \Rightarrow e'_1[x \mapsto v_2], \sigma'' \downarrow v_1, \sigma''' \wedge M_3 \tilde{v}_1 \equiv v_1 \wedge M_3 \tilde{\sigma}''' \equiv \sigma'''.$$

Since M satisfies ϕ_1 , ϕ_2 and ϕ_3 , we obtain from E-APP and the induction steps above that $e_1 e_2, \sigma \downarrow v_1, \sigma'''$ with $M\tilde{v}_1 \equiv v_1$ and $M\tilde{\sigma}''' \equiv \sigma'''$.

Case SE-IF

For all mappings M such that $M(\phi_1 \wedge \phi_2 \wedge \tilde{v}_1)$, we need to demonstrate that **if** e_1 **then** e_2 **else** $e_3, \sigma \downarrow v_2, \sigma''$ with $M\tilde{v}_2 = v_2$ and $M\tilde{\sigma}'' = \sigma''$.

From the induction hypothesis, we obtain the following.

$\forall M_1. M_1\phi_1 \Rightarrow e_1, \sigma \downarrow v_1, \sigma' \wedge M_1\tilde{v}_1 \equiv v_1 \wedge M_1\tilde{\sigma}' \equiv \sigma'$ and

$\forall M_2. M_2\phi_2 \Rightarrow e_2, \sigma' \downarrow v_2, \sigma'' \wedge M_2\tilde{v}_2 \equiv v_2 \wedge M_2\tilde{\sigma}'' \equiv \sigma''$.

Since M satisfies ϕ_1, ϕ_2 and $\tilde{v}_1$, we know that $v_1 = \text{True}$.

From E-IFTRUE and the induction steps above, we obtain that

if e_1 **then** e_2 **else** $e_3, \sigma \downarrow v_2, \sigma''$ with $M\tilde{v}_2 = v_2$ and $M\tilde{\sigma}'' = \sigma''$.

For all mappings M such that $M(\phi_1 \wedge \phi_3 \wedge \neg\tilde{v}_1)$, we need to demonstrate that **if** e_1 **then** e_2 **else** $e_3, \sigma \downarrow v_3, \sigma''$ with $M\tilde{v}_3 = v_3$ and $M\tilde{\sigma}'' = \sigma''$.

From the induction hypothesis, we obtain the following.

$\forall M_1. M_1\phi_1 \Rightarrow e_1, \sigma \downarrow v_1, \sigma' \wedge M_1\tilde{v}_1 \equiv v_1 \wedge M_1\tilde{\sigma}' \equiv \sigma'$ and

$\forall M_3. M_3\phi_3 \Rightarrow e_3, \sigma' \downarrow v_3, \sigma'' \wedge M_3\tilde{v}_3 \equiv v_3 \wedge M_3\tilde{\sigma}'' \equiv \sigma''$.

Since M satisfies ϕ_1, ϕ_3 and $\neg\tilde{v}_1$, we know that $v_1 = \text{False}$.

From E-IFFALSE and the induction steps above, we obtain that

if e_1 **then** e_2 **else** $e_3, \sigma \downarrow v_3, \sigma''$ with $M\tilde{v}_3 = v_3$ and $M\tilde{\sigma}'' = \sigma''$.

Case SE-REF

For all mappings M such that $M\phi$, we need to demonstrate that

ref $e, \sigma \downarrow l, \sigma'[l \mapsto v]$ with $Ml \equiv l$ and $M\tilde{\sigma}'[l \mapsto \tilde{v}] \equiv \sigma'[l \mapsto v]$.

From the induction hypothesis, we obtain the following.

$\forall M_1. M_1\phi \Rightarrow e, \sigma \downarrow v, \sigma' \wedge M_1\tilde{v} \equiv v \wedge M_1\tilde{\sigma}' \equiv \sigma'$.

Since M satisfies ϕ , we obtain from E-REF and the induction steps above that **ref** $e, \sigma \downarrow l, \sigma'[l \mapsto v]$.

We assume that the assignment of location references happens in a deterministic manner, and that we can therefore conclude that exactly the same l is used in both cases. Since l cannot contain any symbols, $Ml \equiv l$ holds trivially.

This, together with $M\tilde{\sigma}' \equiv \sigma'$ obtained from the induction hypothesis, we can conclude that $M\tilde{\sigma}'[l \mapsto \tilde{v}] \equiv \sigma'[l \mapsto v]$.

Case SE-DEREF

For all mappings M such that $M\phi$, we need to demonstrate that $!e, \sigma \downarrow \sigma'(l), \sigma'$ with $M\tilde{\sigma}'(l) \equiv \sigma'(l)$ and $M\tilde{\sigma}' \equiv \sigma'$.

From the induction hypothesis, we obtain the following.

$\forall M_1. M_1\phi \Rightarrow e, \sigma \downarrow l, \sigma' \wedge M_1l \equiv l \wedge M_1\tilde{\sigma}' \equiv \sigma'$.

Since M satisfies ϕ , we obtain from E-DEREF and the induction step above that $!e, \sigma \downarrow \sigma'(l), \sigma'$ with $M\tilde{\sigma}'(l) \equiv \sigma'(l)$ and $M\tilde{\sigma}' \equiv \sigma'$.

Case SE-ASSIGN

For all mappings M such that $M(\phi_1 \wedge \phi_2)$, we need to demonstrate that $e_1 := e_2, \sigma \downarrow \langle \rangle, \sigma''[l \mapsto v_2]$ with $M\langle \rangle \equiv \langle \rangle$, which holds true trivially, and $M\tilde{\sigma}''[l \mapsto \tilde{v}_2] \equiv \sigma''[l \mapsto v_2]$.

From the induction hypothesis, we obtain the following.

$$\begin{aligned} \forall M_1. M_1\phi_1 \Rightarrow e_1, \sigma \downarrow l, \sigma' \wedge M_1l \equiv l \wedge M_1\tilde{\sigma}' \equiv \sigma' \text{ and} \\ \forall M_2. M_2\phi_2 \Rightarrow e_2, \sigma' \downarrow v_2, \sigma'' \wedge M_2\tilde{v}_2 \equiv v_2 \wedge M_2\tilde{\sigma}'' \equiv \sigma'' \end{aligned}$$

Since M satisfies both ϕ_1 and ϕ_2 , we obtain from E-ASSIGN and the induction steps above that $e_1 := e_2, \sigma \downarrow \langle \rangle, \sigma''[l \mapsto v_2]$ with $M\tilde{\sigma}''[l \mapsto \tilde{v}_2] \equiv \sigma''[l \mapsto v_2]$.

Case SE-EDIT

For all mappings M such that $M\phi$, we need to demonstrate that $\Box e, \sigma \downarrow \Box v, \sigma'$ with $M\Box \tilde{v} \equiv \Box v$ and $M\tilde{\sigma}' \equiv \sigma'$.

From the induction hypothesis, we obtain the following.

$$\forall M_1. M_1\phi \Rightarrow e, \sigma \downarrow v, \sigma' \wedge M_1\tilde{v} \equiv v \wedge M_1\tilde{\sigma}' \equiv \sigma'.$$

Since M satisfies ϕ , we obtain from E-EDIT and the induction step above that $\Box e, \sigma \downarrow \Box v, \sigma'$ with $M\Box \tilde{v} \equiv \Box v$ and $M\tilde{\sigma}' \equiv \sigma'$.

Case SE-UPDATE

For all mappings M such that $M\phi$, we need to demonstrate that $\blacksquare e, \sigma \downarrow \blacksquare l, \sigma'$ with $M\blacksquare l \equiv \blacksquare l$ and $M\tilde{\sigma}' \equiv \sigma'$.

From the induction hypothesis, we obtain the following.

$$\forall M_1. M_1\phi \Rightarrow e, \sigma \downarrow l, \sigma' \wedge M_1l \equiv l \wedge M_1\tilde{\sigma}' \equiv \sigma'.$$

Since M satisfies ϕ , we obtain from E-UPDATE and the induction step above that $\blacksquare e, \sigma \downarrow \blacksquare l, \sigma'$ with $M\blacksquare l \equiv \blacksquare l$ and $M\tilde{\sigma}' \equiv \sigma'$.

Case SE-THEN

For all mappings M such that $M\phi$, we need to demonstrate that $e_1 \blacktriangleright e_2, \sigma \downarrow t_1 \blacktriangleright e_2, \sigma'$ with $M\tilde{t}_1 \blacktriangleright \tilde{e}_2 \equiv t_1 \blacktriangleright e_2$ and $M\tilde{\sigma}' \equiv \sigma'$.

From the induction hypothesis, we obtain the following.

$$\forall M_1. M_1\phi \Rightarrow e, \sigma \downarrow t_1, \sigma' \wedge M_1\tilde{t}_1 \equiv t_1 \wedge M_1\tilde{\sigma}' \equiv \sigma'.$$

Since M satisfies ϕ , we obtain from E-THEN and the induction step above that $e_1 \blacktriangleright e_2, \sigma \downarrow t_1 \blacktriangleright e_2, \sigma'$ with $M\tilde{t}_1 \blacktriangleright \tilde{e}_2 \equiv t_1 \blacktriangleright e_2$ and $M\tilde{\sigma}' \equiv \sigma'$.

Case SE-NEXT

For all mappings M such that $M\phi$, we need to demonstrate that $e_1 \triangleright e_2, \sigma \downarrow t_1 \triangleright e_2, \sigma'$ with $M\tilde{t}_1 \triangleright \tilde{e}_2 \equiv t_1 \triangleright e_2$ and $M\tilde{\sigma}' \equiv \sigma'$.

From the induction hypothesis, we obtain the following.

$$\forall M_1. M_1\phi \Rightarrow e, \sigma \downarrow t_1, \sigma' \wedge M_1\tilde{t}_1 \equiv t_1 \wedge M_1\tilde{\sigma}' \equiv \sigma'.$$

Since M satisfies ϕ , we obtain from E-NEXT and the induction step above that $e_1 \triangleright e_2, \sigma \downarrow t_1 \triangleright e_2, \sigma'$ with $M\tilde{t}_1 \triangleright \tilde{e}_2 \equiv t_1 \triangleright e_2$ and $M\tilde{\sigma}' \equiv \sigma'$.

Case SE-OR

For all mappings M such that $M(\phi_1 \wedge \phi_2)$, we need to demonstrate that $e_1 \blacklozenge e_2, \sigma \downarrow t_1 \blacklozenge t_2, \sigma''$ with $M\tilde{t}_1 \blacklozenge \tilde{t}_2 \equiv t_1 \blacklozenge t_2$ and $M\tilde{\sigma}'' \equiv \sigma''$.

From the induction hypothesis, we obtain the following.

$\forall M_1. M_1\phi_1 \Rightarrow e_1, \sigma \downarrow t_1, \sigma' \wedge M_1\tilde{t}_1 \equiv t_1 \wedge M_1\tilde{\sigma}' \equiv \sigma'$ and

$\forall M_2. M_2\phi_2 \Rightarrow e_2, \sigma' \downarrow t_2, \sigma'' \wedge M_2\tilde{t}_2 \equiv t_2 \wedge M_2\tilde{\sigma}'' \equiv \sigma''$

Since M satisfies both ϕ_1 and ϕ_2 , we obtain from E-OR and the induction steps above that $e_1 \blacklozenge e_2, \sigma \downarrow t_1 \blacklozenge t_2, \sigma''$ with $M\tilde{t}_1 \blacklozenge \tilde{t}_2 \equiv t_1 \blacklozenge t_2$ and $M\tilde{\sigma}'' \equiv \sigma''$.

Case SE-AND

For all mappings M such that $M(\phi_1 \wedge \phi_2)$, we need to demonstrate that $e_1 \bowtie e_2, \sigma \downarrow t_1 \bowtie t_2, \sigma''$ with $M\tilde{t}_1 \bowtie \tilde{t}_2 \equiv t_1 \bowtie t_2$ and $M\tilde{\sigma}'' \equiv \sigma''$.

From the induction hypothesis, we obtain the following.

$\forall M_1. M_1\phi_1 \Rightarrow e_1, \sigma \downarrow t_1, \sigma' \wedge M_1\tilde{t}_1 \equiv t_1 \wedge M_1\tilde{\sigma}' \equiv \sigma'$ and

$\forall M_2. M_2\phi_2 \Rightarrow e_2, \sigma' \downarrow t_2, \sigma'' \wedge M_2\tilde{t}_2 \equiv t_2 \wedge M_2\tilde{\sigma}'' \equiv \sigma''$

Since M satisfies both ϕ_1 and ϕ_2 , we obtain from E-AND and the induction steps above that $e_1 \bowtie e_2, \sigma \downarrow t_1 \bowtie t_2, \sigma''$ with $M\tilde{t}_1 \bowtie \tilde{t}_2 \equiv t_1 \bowtie t_2$ and $M\tilde{\sigma}'' \equiv \sigma''$.

□

D.1.2 Proof of soundness of symbolic striding semantics

Proof: We prove Lemma 6.5.4 by induction over the derivation $t, \sigma \rightsquigarrow \tilde{t}, \tilde{\sigma}, \phi$.

Case SS-THENSTAY, SS-THENFAIL

For all mappings M such that $M\phi$ we need to demonstrate that $t_1 \blacktriangleright e_2, \sigma \mapsto t'_1 \blacktriangleright e_2, \sigma'$ with $M\tilde{t}'_1 \blacktriangleright e_2 \equiv t'_1 \blacktriangleright e_2$ and $M\tilde{\sigma}' \equiv \sigma'$.

From the induction hypothesis, we obtain the following.

$\forall M_1. M_1\phi \Rightarrow t_1, \sigma \mapsto t'_1, \sigma' \wedge M_1\tilde{t}'_1 \equiv t'_1 \wedge M_1\tilde{\sigma}' \equiv \sigma'$.

Since M satisfies ϕ , we obtain from S-THENSTAY and S-THENFAIL respectively, and the induction step above that $t_1 \blacktriangleright e_2, \sigma \mapsto t'_1 \blacktriangleright e_2, \sigma'$ with $M\tilde{t}'_1 \blacktriangleright e_2 \equiv t'_1 \blacktriangleright e_2$ and $M\tilde{\sigma}' \equiv \sigma'$.

Case SS-THENCONT

For all mappings M such that $M\phi_1 \wedge M\phi_2$ we need to demonstrate that $t_1 \blacktriangleright e_2, \sigma \mapsto t_2, \sigma''$ with $M\tilde{t}_2 \equiv t_2$ and $M\tilde{\sigma}'' \equiv \sigma''$.

From the induction hypothesis, we obtain the following.

$\forall M_1. M_1\phi_1 \Rightarrow t_1, \sigma \mapsto t'_1, \sigma' \Rightarrow M_1\tilde{t}'_1 \equiv t'_1 \wedge M_1\tilde{\sigma}' \equiv \sigma'$.

From Lemma 6.5.5 we know that

$$\forall M_2. M_2 \phi_2 \Rightarrow e_2 v_1, \sigma' \downarrow t_2, \sigma'' \text{ and } M_2 \tilde{t}_2 \equiv t_2 \wedge M_2 \tilde{\sigma}'' \equiv \sigma''.$$

Since M satisfies both ϕ_1 and ϕ_2 , we obtain from S-THENCONT, the induction step and application of Lemma 6.5.5 above that $t_1 \blacktriangleright e_2, \sigma \mapsto t_2, \sigma''$ with $M \tilde{t}_2 \equiv t_2$ and $M \tilde{\sigma}'' \equiv \sigma''$.

Case SS-ORLEFT

For all mappings M such that $M\phi$ we have to demonstrate that $t_1 \blacklozenge t_2, \sigma \mapsto t'_1, \sigma'$ with $M \tilde{t}'_1 \equiv t'_1$ and $M \tilde{\sigma}' \equiv \sigma'$.

From the induction hypothesis, we obtain the following.

$$\forall M_1. M_1 \phi \Rightarrow t_1, \sigma \mapsto t'_1, \sigma' \text{ and } M_1 \tilde{t}'_1 \equiv t'_1 \wedge M_1 \tilde{\sigma}' \equiv \sigma'.$$

Since M satisfies ϕ , we obtain from S-ORLEFT and the induction step above that $t_1 \blacklozenge t_2, \sigma \mapsto t'_1, \sigma'$ with $M \tilde{t}'_1 \equiv t'_1$ and $M \tilde{\sigma}' \equiv \sigma'$.

Case SS-ORRIGHT

For all mappings M such that $M(\phi_1 \wedge \phi_2)$ we need to demonstrate that $t_1 \blacklozenge t_2, \sigma \mapsto t'_2, \sigma''$ with $M \tilde{t}'_2 \equiv t'_2$ and $M \tilde{\sigma}'' \equiv \sigma''$.

From the induction hypothesis, we obtain the following.

$$\forall M_1. M_1 \phi_1 \Rightarrow t_1, \sigma \mapsto t'_1, \sigma' \wedge M_1 \tilde{t}'_1 \equiv t'_1 \wedge M_1 \tilde{\sigma}' \equiv \sigma' \text{ and}$$

$$\forall M_2. M_2 \phi_2 \Rightarrow t_2, \sigma' \mapsto t'_2, \sigma'' \wedge M_2 \tilde{t}'_2 \equiv t'_2 \wedge M_2 \tilde{\sigma}'' \equiv \sigma''.$$

Since M satisfies both ϕ_1 and ϕ_2 , and from the premise we have that $\mathcal{V}(\tilde{t}', \tilde{\sigma}') = \perp$, we obtain from S-ORRIGHT and the induction steps above that $t_1 \blacklozenge t_2, \sigma \mapsto t'_2, \sigma''$ with $M \tilde{t}'_2 \equiv t'_2$ and $M \tilde{\sigma}'' \equiv \sigma''$.

Case SS-ORNONE

For all mappings M such that $M(\phi_1 \wedge \phi_2)$ we need to demonstrate that $t_1 \blacklozenge t_2, \sigma \mapsto t'_1 \blacklozenge t'_2, \sigma''$ with $M \tilde{t}'_1 \blacklozenge \tilde{t}'_2 \equiv t'_1 \blacklozenge t'_2$ and $M \tilde{\sigma}'' \equiv \sigma''$.

From the induction hypothesis, we obtain the following.

$$\forall M_1. M_1 \phi_1 \Rightarrow t_1, \sigma \mapsto t'_1, \sigma' \wedge M_1 \tilde{t}'_1 \equiv t'_1 \wedge M_1 \tilde{\sigma}' \equiv \sigma' \text{ and}$$

$$\forall M_2. M_2 \phi_2 \Rightarrow t_2, \sigma' \mapsto t'_2, \sigma'' \wedge M_2 \tilde{t}'_2 \equiv t'_2 \wedge M_2 \tilde{\sigma}'' \equiv \sigma''.$$

Since M satisfies both ϕ_1 and ϕ_2 , we obtain from S-ORNONE and the induction steps above that $t_1 \blacklozenge t_2, \sigma \mapsto t'_1 \blacklozenge t'_2, \sigma''$ with $M \tilde{t}'_1 \blacklozenge \tilde{t}'_2 \equiv t'_1 \blacklozenge t'_2$ and $M \tilde{\sigma}'' \equiv \sigma''$.

Case SS-EDIT

For all mappings M , we need to demonstrate that $\Box v, \sigma \mapsto \Box v, \sigma$ with $M \Box v \equiv \Box v$ and $M \sigma \equiv \sigma$. This follows trivially from S-EDIT.

Case SS-FILL

For all mappings M , we need to demonstrate that $\boxtimes \beta, \sigma \mapsto \boxtimes \beta, \sigma$ with $M \boxtimes \beta \equiv \boxtimes \beta$ and $M \sigma \equiv \sigma$. This follows trivially from S-FILL.

Case SS-UPDATE

For all mappings M , we need to demonstrate that $\blacksquare l, \sigma \mapsto \blacksquare l, \sigma$ with $M \blacksquare l \equiv \blacksquare l$ and $M\sigma \equiv \sigma$. This follows trivially from S-UPDATE.

Case SS-FAIL

For all mappings M , we need to demonstrate that $\downarrow, \sigma \mapsto \downarrow, \sigma$ with $M \downarrow \equiv \downarrow$ and $M\sigma \equiv \sigma$. This follows trivially from S-FAIL.

Case SS-XOR

For all mappings M , we need to demonstrate that $e_1 \diamond e_2, \sigma \mapsto e_1 \diamond e_2, \sigma$ with $Me_1 \diamond e_2 \equiv e_1 \diamond e_2$ and $M\sigma \equiv \sigma$. This follows trivially from S-XOR.

Case SS-NEXT

For all mappings M such that $M\phi$, we need to demonstrate that $t_1 \triangleright e_2, \sigma \mapsto t'_1 \triangleright e_2, \sigma'$ with $M\tilde{t}'_1 \triangleright e_2 \equiv t'_1 \triangleright e_2$ and $M\tilde{\sigma}' \equiv \sigma'$.

From the induction hypothesis, we obtain the following.

$$\forall M_1. M_1\phi \Rightarrow t_1, \sigma \mapsto t'_1, \sigma' \wedge M_1\tilde{t}'_1 \equiv t'_1 \wedge M_1\tilde{\sigma}' \equiv \sigma'.$$

Since M satisfies ϕ , we obtain from S-NEXT and the induction step above that $t_1 \triangleright e_2, \sigma \mapsto t'_1 \triangleright e_2, \sigma'$ with $M\tilde{t}'_1 \triangleright e_2 \equiv t'_1 \triangleright e_2$ and $M\tilde{\sigma}' \equiv \sigma'$.

Case SS-AND

For all mappings M such that $M(\phi_1 \wedge \phi_2)$ we need to demonstrate that $t_1 \bowtie t_2, \sigma \mapsto t'_1 \bowtie t'_2, \sigma''$ with $M\tilde{t}'_1 \bowtie \tilde{t}'_2 \equiv t'_1 \bowtie t'_2$ and $M\tilde{\sigma}'' \equiv \sigma''$.

From the induction hypothesis, we obtain the following.

$$\forall M_1. M_1\phi_1 \Rightarrow t_1, \sigma \mapsto t'_1, \sigma' \text{ and } M_1\tilde{t}'_1 \equiv t'_1 \wedge M_1\tilde{\sigma}' \equiv \sigma' \text{ and}$$

$$\forall M_2. M_2\phi_2 \Rightarrow t_2, \sigma' \mapsto t'_2, \sigma'' \text{ and } M_2\tilde{t}'_2 \equiv t'_2 \wedge M_2\tilde{\sigma}'' \equiv \sigma''.$$

Since M satisfies both ϕ_1 and ϕ_2 , we obtain from S-AND and the induction steps above that $t_1 \bowtie t_2, \sigma \mapsto t'_1 \bowtie t'_2, \sigma''$ with $M\tilde{t}'_1 \bowtie \tilde{t}'_2 \equiv t'_1 \bowtie t'_2$ and $M\tilde{\sigma}'' \equiv \sigma''$.

□

D.1.3 Proof of soundness of symbolic normalisation semantics

Proof: We prove Lemma 6.5.3 by induction over the derivation $e, \sigma \Downarrow \tilde{t}, \tilde{\sigma}, \phi$.

The base case is when the SN-Done rule applies. Provided that $M(\phi_1 \wedge \phi_2)$, we need to demonstrate that $e, \sigma \Downarrow t, \sigma'$ with $M\tilde{t} \equiv t$ and $M\tilde{\sigma}' \equiv \sigma'$.

By Lemma 6.5.5 and 6.5.4, we know that

$$\forall M_1. M_1\phi_1 \Rightarrow e, \sigma \Downarrow t, \sigma' \wedge M_1\tilde{t} \equiv t \wedge M_1\tilde{\sigma}' \equiv \sigma' \text{ and}$$

$$\forall M_2. M_2\phi_2 \Rightarrow t, \sigma' \mapsto t', \sigma'' \wedge M_2\tilde{t}' \equiv t' \wedge M_2\tilde{\sigma}'' \equiv \sigma''.$$

Since M satisfies both ϕ_1 and ϕ_2 , we have $e, \sigma \Downarrow t, \sigma'$ with $M\tilde{\sigma}' \equiv \sigma'$.

The induction step is when SN-REPEAT applies. In this case, for all mappings M such that $M(\phi_1 \wedge \phi_2 \wedge \phi_3)$, we need to demonstrate that $e, \sigma \Downarrow t'', \sigma'''$ with $M\tilde{t}'' \equiv t''$ and $M\tilde{\sigma}''' \equiv \sigma'''$.

Again by Lemma 6.5.5 and 6.5.4, we know that
 $\forall M_1. M_1\phi_1 \Rightarrow e, \sigma \Downarrow t, \sigma' \wedge M_1\tilde{t} \equiv t \wedge M_1\tilde{\sigma}' \equiv \sigma'$ and
 $\forall M_2. M_2\phi_2 \Rightarrow t, \sigma' \mapsto t', \sigma'' \wedge M_2\tilde{t}' \equiv t' \wedge M_2\tilde{\sigma}'' \equiv \sigma''$.

Furthermore, we know by applying the induction hypothesis that
 $\forall M_3. M_3\phi_3 \Rightarrow t', \sigma'' \Downarrow t'', \sigma''' \wedge M_3\tilde{t}'' \equiv t'' \wedge M_3\tilde{\sigma}''' \equiv \sigma'''$.

Since M satisfies ϕ_1, ϕ_2 and ϕ_3 , we obtain from N-REPEAT, the application of lemmas and the induction step above that $e, \sigma \Downarrow t'', \sigma'''$ with $M\tilde{t}'' \equiv t''$ and $M\tilde{\sigma}''' \equiv \sigma'''$. $\square$

D.1.4 Proof of soundness of symbolic handling semantics

Proof: We prove Lemma 6.5.2 by induction over the derivation $t, \sigma \rightsquigarrow \tilde{t}, \tilde{\sigma}, \tilde{t}, \phi$.

Case SH-CHANGE

For all mappings M , we need to demonstrate that $\Box v, \sigma \xrightarrow{M_S} \Box Ms, \sigma$ with $M\Box s \equiv \Box Ms$ and $M\sigma \equiv \sigma$.

This follows trivially from H-CHANGE.

Case SH-FILL

For all mappings M , we need to demonstrate that $\Box\beta, \sigma \xrightarrow{M_S} \Box Ms, \sigma$ with $M\Box s \equiv \Box Ms$ and $M\sigma \equiv \sigma$.

This follows trivially from H-FILL.

Case SH-UPDATE

For all mappings M , we need to demonstrate that

$\blacksquare l, \sigma \xrightarrow{M_S} \blacksquare l, \sigma[l \mapsto Ms]$ with $M\blacksquare l \equiv \blacksquare l$ and $M\sigma[l \mapsto s] \equiv \sigma[l \mapsto Ms]$.

$\blacksquare l, \sigma \xrightarrow{M_S} \blacksquare l, \sigma[l \mapsto Ms]$ follows trivially from H-UPDATE. $M\blacksquare l \equiv \blacksquare l$ follows trivially, since locations cannot contain symbols. $M\sigma[l \mapsto s] \equiv \sigma[l \mapsto Ms]$ follows trivially.

Case SH-NEXT

For all mappings M such that $M\phi_1$, we need to demonstrate that

$t_1 \triangleright e_2, \sigma \xrightarrow{M\tilde{t}} t'_1 \triangleright e_2, \sigma'$ with $M\tilde{t}'_1 \triangleright e_2 \equiv t'_1 \triangleright e_2$ and $M\tilde{\sigma}' \equiv \sigma'$.

By the induction hypothesis we obtain the following.

$\forall M_1. M_1\phi_1 \Rightarrow t_1, \sigma \xrightarrow{M_1\tilde{t}} t'_1, \sigma' \wedge M_1\tilde{t}'_1 \equiv t'_1 \wedge M_1\tilde{\sigma}' \equiv \sigma'$

Since M satisfies ϕ_1 , we obtain from H-PASSNEXT and the induction step above that $t_1 \triangleright e_2, \sigma \xrightarrow{M\tilde{t}} t'_1 \triangleright e_2, \sigma'$ with $M\tilde{t}'_1 \triangleright e_2 \equiv t'_1 \triangleright e_2$ and $M\tilde{\sigma}' \equiv \sigma'$.

For all mappings M such that $M\phi_2$, we need to demonstrate that

$$t_1 \triangleright e_2, \sigma \xrightarrow{C} t_2, \sigma' \text{ with } M\tilde{t}_2 \equiv t_2 \text{ and } M\tilde{\sigma}' \equiv \sigma'.$$

From Lemma 6.5.3 we obtain that $\forall M_1. M_1\phi \Rightarrow e_2v_1, \sigma \Downarrow t_2, \sigma' \wedge M\tilde{t}_2 \equiv t_2 \wedge M\tilde{\sigma}' \equiv \sigma'$.

This together with H-NEXT gives us exactly what we need to prove this case.

Case SH-PASSNEXT

For all mappings M such that $M\phi$, we need to demonstrate that

$$t_1 \triangleright e_2, \sigma \xrightarrow{M\tilde{t}} t'_1 \triangleright e_2, \sigma' \text{ with } M\tilde{t}'_1 \triangleright e_2 \equiv t'_1 \triangleright e_2 \text{ and } M\tilde{\sigma}' \equiv \sigma'.$$

By the induction hypothesis we obtain the following.

$$\forall M_1. M_1\phi_1 \Rightarrow t_1, \sigma \xrightarrow{M_1\tilde{t}} t'_1, \sigma' \wedge M_1\tilde{t}'_1 \equiv t'_1 \wedge M_1\tilde{\sigma}' \equiv \sigma'$$

Since M satisfies ϕ , we obtain from H-PASSNEXT and the induction step above that $t_1 \triangleright e_2, \sigma \xrightarrow{M\tilde{t}} t'_1 \triangleright e_2, \sigma'$ with $M\tilde{t}'_1 \triangleright e_2 \equiv t'_1 \triangleright e_2$ and $M\tilde{\sigma}' \equiv \sigma'$.

Case SH-PASSNEXTFAIL

For all mappings M such that $M\phi$, we need to demonstrate that

$$t_1 \triangleright e_2, \sigma \xrightarrow{M\tilde{t}} t'_1 \triangleright e_2, \sigma' \text{ with } M\tilde{t}'_1 \triangleright e_2 \equiv t'_1 \triangleright e_2 \text{ and } M\tilde{\sigma}' \equiv \sigma'.$$

By the induction hypothesis we obtain the following.

$$\forall M_1. M_1\phi_1 \Rightarrow t_1, \sigma \xrightarrow{M_1\tilde{t}} t'_1, \sigma' \wedge M_1\tilde{t}'_1 \equiv t'_1 \wedge M_1\tilde{\sigma}' \equiv \sigma'.$$

Since M satisfies ϕ and from the premise of SH-PASSNEXTFAIL we have $\mathcal{F}(\tilde{t}_2, \tilde{\sigma}'')$, we obtain from H-PASSNEXTFAIL and the induction step above that $t_1 \triangleright e_2, \sigma \xrightarrow{M\tilde{t}} t'_1 \triangleright e_2, \sigma'$ with $M\tilde{t}'_1 \triangleright e_2 \equiv t'_1 \triangleright e_2$ and $M\tilde{\sigma}' \equiv \sigma'$.

Case SH-PASSTHEN

For all mappings M such that $M\phi$, we need to demonstrate that

$$t_1 \blacktriangleright e_2, \sigma \xrightarrow{M\tilde{t}} t'_1 \blacktriangleright e_2, \sigma' \text{ with } M\tilde{t}'_1 \blacktriangleright e_2 \equiv t'_1 \blacktriangleright e_2 \text{ and } M\tilde{\sigma}' \equiv \sigma'.$$

By the induction hypothesis we obtain the following.

$$\forall M_1. M_1\phi_1 \Rightarrow t_1, \sigma \xrightarrow{M_1\tilde{t}} t'_1, \sigma' \wedge M_1\tilde{t}'_1 \equiv t'_1 \wedge M_1\tilde{\sigma}' \equiv \sigma'$$

Since M satisfies ϕ , we obtain from H-PASSTHEN and the induction step above that $t_1 \blacktriangleright e_2, \sigma \xrightarrow{M\tilde{t}} t'_1 \blacktriangleright e_2, \sigma'$ with $M\tilde{t}'_1 \blacktriangleright e_2 \equiv t'_1 \blacktriangleright e_2$ and $M\tilde{\sigma}' \equiv \sigma'$.

Case SH-PICK

We have that $M\phi_1$ and/or $M\phi_2$. In the first case, the proof is identical to the SH-PickLeft rule. In the second case, the proof is identical to the SH-PickRight rule.

Case SH-PICKLEFT

For all mappings M such that $M\phi_1$, we need to demonstrate that

$$e_1 \diamond e_2, \sigma \xrightarrow{L} t_1, \sigma' \text{ with } M\tilde{t}_1 \equiv t_1 \text{ and } M\tilde{\sigma}' \equiv \sigma'.$$

From Lemma 6.5.3 we obtain that $\forall M_1. M_1\phi \Rightarrow e_1, \sigma \Downarrow t_1, \sigma' \wedge M\tilde{t}_1 \equiv t_1 \wedge M\tilde{\sigma}' \equiv \sigma'$.

Since M satisfies ϕ_1 , we obtain from H-PICKLEFT and the application of Lemma 6.5.3 above that $e_1 \diamond e_2, \sigma \xrightarrow{L} t_1, \sigma'$ with $M\tilde{t}_1 \equiv t_1$ and $M\tilde{\sigma}' \equiv \sigma'$.

Case SH-PICKRIGHT

For all mappings M such that $M\phi_2$, we need to demonstrate that

$$e_1 \diamond e_2, \sigma \xrightarrow{R} t_2, \sigma' \text{ with } M\tilde{t}_2 \equiv t_2 \text{ and } M\tilde{\sigma}_2 \equiv \sigma'.$$

From Lemma 6.5.3 we obtain that $\forall M_1. M_1\phi \Rightarrow e_2, \sigma \Downarrow t_2, \sigma' \wedge M\tilde{t}_2 \equiv t_2 \wedge M\tilde{\sigma}' \equiv \sigma'$.

Since M satisfies ϕ_2 , we obtain from H-PICKRIGHT and the application of Lemma 6.5.3 above that $e_1 \diamond e_2, \sigma \xrightarrow{R} t_2, \sigma'$ with $M\tilde{t}_2 \equiv t_2$ and $M\tilde{\sigma}_2 \equiv \sigma'$.

Case SH-AND

For all mappings M such that $M\phi_1$, we need to demonstrate that

$$t_1 \bowtie t_2, \sigma \xrightarrow{MF\tilde{I}} t'_1 \bowtie t_2, \sigma' \text{ with } M\tilde{t}'_1 \bowtie t_2 \equiv t'_1 \bowtie t_2 \text{ and } M\tilde{\sigma}' \equiv \sigma'.$$

By the induction hypothesis we obtain the following.

$$\forall M_1. M_1\phi_1 \Rightarrow t_1, \sigma \xrightarrow{M_1\tilde{I}} t'_1, \sigma' \wedge M_1\tilde{t}'_1 \equiv t'_1 \wedge M_1\tilde{\sigma}' \equiv \sigma'.$$

Since M satisfies ϕ_1 , we obtain from H-FIRSTAND and the induction step above that $t_1 \bowtie t_2, \sigma \xrightarrow{MF\tilde{I}} t'_1 \bowtie t_2, \sigma'$ with $M\tilde{t}'_1 \bowtie t_2 \equiv t'_1 \bowtie t_2$ and $M\tilde{\sigma}' \equiv \sigma'$.

For all mappings M such that $M\phi_2$, we need to demonstrate that

$$t_1 \bowtie t_2, \sigma \xrightarrow{MS\tilde{I}} t_1 \bowtie t'_2, \sigma' \text{ with } Mt_1 \bowtie \tilde{t}'_2 \equiv t_1 \bowtie t'_2 \text{ and } M\tilde{\sigma}' \equiv \sigma'.$$

By the induction hypothesis we obtain the following.

$$\forall M_1. M_1\phi_2 \Rightarrow t_2, \tilde{\sigma} \xrightarrow{M_1\tilde{I}} t'_2, \sigma' \wedge M_1\tilde{t}'_2 \equiv t'_2 \wedge M_1\tilde{\sigma}' \equiv \sigma'$$

Since M satisfies ϕ_2 , we obtain from H-SECONDAND and the induction step above that $t_1 \bowtie t_2, \sigma \xrightarrow{MS\tilde{I}} t_1 \bowtie t'_2, \sigma'$ with $Mt_1 \bowtie \tilde{t}'_2 \equiv t_1 \bowtie t'_2$ and $M\tilde{\sigma}' \equiv \sigma'$.

Case SH-OR

This case is proven in the same way as SH-AND.

□

D.1.5 Proof of soundness of symbolic interacting semantics

Proof: We prove Lemma 6.5.1 by induction on $\tilde{t}, \tilde{\sigma} \approx \overline{\tilde{t}', \tilde{\sigma}', \tilde{t}, \phi}$. There is only one rule that applies, namely SI-HANDLE.

Provided that $M(\phi_1 \wedge \phi_2)$, we need to demonstrate that $t, \sigma \xRightarrow{M\tilde{t}} t'', \sigma''$ with $M\tilde{t}'' \equiv t''$ and $M\tilde{\sigma}'' \equiv \sigma''$.

Lemma 6.5.2 and Lemma 6.5.3 respectively give us that

$$\begin{aligned} \forall M_1. M_1\phi_1 \Rightarrow t, \sigma \xrightarrow{M_1\tilde{t}} t', \sigma' \wedge M_1\tilde{t}' \equiv t' \wedge M_1\tilde{\sigma}' \equiv \sigma' \text{ and} \\ \forall M_2. M_2\phi_2 \Rightarrow t', \sigma' \Downarrow t'', \sigma'' \wedge M_2\tilde{t}'' \equiv t'' \wedge M_2\tilde{\sigma}'' \equiv \sigma''. \end{aligned}$$

Since M satisfies both ϕ_1 and ϕ_2 , we obtain exactly what we need to prove, namely $t, \sigma \xRightarrow{\tilde{t}} t'', \sigma''$ $M\tilde{t}'' \equiv t''$ and $M\tilde{\sigma}'' \equiv \sigma''$.

□

D.2 Completeness proofs

D.2.1 Proof of completeness of the symbolic handling semantics

Proof: We prove Lemma 6.5.8 by induction over the derivation $t, \sigma \xrightarrow{i} t', \sigma'$.

Case H-CHANGE

By the SH-Change rule, we have $\Box v, \sigma \rightsquigarrow \Box s, \tilde{\sigma}, s, \text{True}$, and $s \sim v'$ holds by definition of input simulation.

Case H-FILL

By the SH-Fill rule, we have $\Box \beta, \sigma \rightsquigarrow \Box s, \tilde{\sigma}, s, \text{True}$, and $s \sim v$ holds by definition of input simulation.

Case H-UPDATE

By the SH-Update rule, we have $\blacksquare l, \sigma \rightsquigarrow \blacksquare l, \tilde{\sigma}[l \mapsto s], s, \text{True}$, and $s \sim v$ holds by definition of input simulation.

Case H-NEXT

By the SH-Next rule, we have $t_1 \triangleright e_2, \sigma \rightsquigarrow \overline{\tilde{t}'_1 \triangleright e_2, \tilde{\sigma}_1, \tilde{l}, \phi_1} \cup \overline{\tilde{t}_2, \tilde{\sigma}_2, C, \phi_2}$, and $C \sim C$ holds by definition of input simulation.

Case H-PASSNEXT

By application of the induction hypothesis, we obtain the following.
For all t_1, σ, i such that $t_1, \sigma \xrightarrow{i} t'_1, \sigma'$ there exists an $\tilde{i} \sim i$ such that $t_1, \sigma \rightsquigarrow \tilde{t}_1, \tilde{\sigma}, \tilde{l}, \phi$. From this we can conclude that there exists a symbolic execution $t_1 \triangleright e_2, \sigma \rightsquigarrow \overline{\tilde{t}_1 \triangleright e_2, \tilde{\sigma}, \tilde{l}, \phi}$, and that $\tilde{i} \sim i$.

Case H-PASSTHEN

By application of the induction hypothesis, we obtain the following.
For all t_1, σ, i such that $t_1, \sigma \xrightarrow{i} t'_1, \sigma'$ there exists an $\tilde{i} \sim i$ such that $t_1, \sigma \rightsquigarrow \tilde{t}_1, \tilde{\sigma}, \tilde{l}, \phi$. From this we can conclude that there exists a symbolic execution $t_1 \blacktriangleright e_2, \sigma \rightsquigarrow \overline{\tilde{t}_1 \blacktriangleright e_2, \tilde{\sigma}, \tilde{l}, \phi}$, and $\tilde{i} \sim i$.

Case H-PICKLEFT

Lemma 6.5.9 gives us the following.
There exists a symbolic execution $e_1, \sigma \rightsquigarrow \overline{\tilde{t}_1, \tilde{\sigma}, \phi_1}$.
There exists a symbolic execution $e_2, \tilde{\sigma} \rightsquigarrow \overline{\tilde{t}_2, \tilde{\sigma}', \phi_2}$.

We can now conclude that a symbolic execution exists. Either by the SH-PICKLEFT rule, in case $\mathcal{F}(\tilde{t}_2, \tilde{\sigma}')$, or by the SH-PICK rule in case $\neg \mathcal{F}(\tilde{t}_2, \tilde{\sigma}')$. We have that $L \sim L$ holds by definition.

Case H-PICKRIGHT

Lemma 6.5.9 gives us the following.

There exists a symbolic execution $e_1, \sigma \Downarrow \overline{t_1, \tilde{\sigma}, \phi_1}$.

There exists a symbolic execution $e_2, \tilde{\sigma} \Downarrow \overline{t_2, \tilde{\sigma}', \phi_2}$.

We can now conclude that a symbolic execution exists. Either by the SH-PICKRIGHT rule, in case $\mathcal{F}(\tilde{t}_1, \tilde{\sigma})$, or by the SH-PICK rule in case $\neg \mathcal{F}(\tilde{t}_1, \tilde{\sigma})$.

We have that $R \sim R$ holds by definition.

Case H-FIRSTOR

By application of the induction hypothesis, we obtain the following.

For all t_1, σ, i such that $t_1, \sigma \xrightarrow{i} t'_1, \sigma'$ there exists an $\tilde{i} \sim i$ such that $t_1, \sigma \rightsquigarrow \tilde{t}_1, \tilde{\sigma}, \tilde{i}, \phi$.

From SH-OR, and the conclusion of the induction hypothesis, we can conclude that there exists a symbolic input, namely $F\tilde{i}$, such that $t_1 \blacklozenge t_2, \sigma \rightsquigarrow \tilde{t}'_1 \blacklozenge t_2, \tilde{\sigma}, F\tilde{i}, \phi$. From $\tilde{i} \sim i$ and by definition of input simulation, we can conclude that $F\tilde{i} \sim Fi$.

Case H-SECONDOR

By application of the induction hypothesis, we obtain the following.

For all t_2, σ, i such that $t_2, \sigma \xrightarrow{i} t'_2, \sigma'$ there exists an $\tilde{i} \sim i$ such that $t_2, \sigma \rightsquigarrow \tilde{t}_2, \tilde{\sigma}, \tilde{i}, \phi$.

From SH-OR, and the induction step above, we can conclude that there exists a symbolic input such that $t_1 \blacklozenge t_2, \sigma \rightsquigarrow \overline{\tilde{t}_1 \blacklozenge t'_2, \tilde{\sigma}', S\tilde{i}, \phi}$, namely $S\tilde{i}$. From $\tilde{i} \sim i$ and by definition of input simulation, we can conclude that $S\tilde{i} \sim Si$.

Case H-FIRSTAND

By application of the induction hypothesis, we obtain the following.

For all t_1, σ, i such that $t_1, \sigma \xrightarrow{i} t'_1, \sigma'$ there exists an $\tilde{i} \sim i$ such that $t_1, \sigma \rightsquigarrow \tilde{t}_1, \tilde{\sigma}, \tilde{i}, \phi$.

From SH-AND, and the conclusion of the induction step above, we can conclude that there exists a symbolic input, namely $F\tilde{i}$ such that $t_1 \bowtie t_2, \sigma \rightsquigarrow \overline{\tilde{t}'_1 \bowtie t_2, \tilde{\sigma}, F\tilde{i}, \phi}$. From $\tilde{i} \sim i$ and by definition of input simulation, we can conclude that $F\tilde{i} \sim Fi$.

Case H-SECONDAND

By application of the induction hypothesis, we obtain the following.

For all t_2, σ, i such that $t_2, \sigma \xrightarrow{i} t'_2, \sigma'$ there exists an $\tilde{i} \sim i$ such that $t_2, \sigma \rightsquigarrow \tilde{t}_2, \tilde{\sigma}, \tilde{i}, \phi$.

From SH-AND, and the conclusion of the induction step above, we can conclude that there exists a symbolic input, namely $S\tilde{i}$ such that $t_1 \bowtie t_2, \sigma \rightsquigarrow t_1 \bowtie \tilde{t}_2, \tilde{\sigma}, S\tilde{i}, \phi$. From $\tilde{i} \sim i$ and by definition of input simulation, we can conclude that $S\tilde{i} \sim Si$.

□

D.2.2 Proof of completeness of the symbolic interaction semantics

Proof: The proof of Theorem 6.5.7 consists of one case, since the interacting semantics consists of one rule, namely

I-HANDLE

$$\frac{t, \sigma \xrightarrow{i} t', \sigma' \quad t', \sigma' \Downarrow t'', \sigma''}{t, \sigma \xRightarrow{i} t'', \sigma''}$$

By Lemma 6.5.8 we obtain the following.

$$t, \sigma \xrightarrow{i} t', \sigma' \Rightarrow \exists \tilde{i}. t, \sigma \rightsquigarrow \tilde{t}, \tilde{\sigma}, \tilde{i}, \phi \wedge \tilde{i} \sim i$$

Then by Lemma 6.5.9 we obtain the following.

$$t', \sigma' \Downarrow t'', \sigma'' \Rightarrow t', \sigma' \Downarrow \tilde{t}', \tilde{\sigma}', \phi'$$

From the above, together with the SI-Handle rule, we can conclude that there exists a symbolic execution $t, \sigma \rightsquigarrow \tilde{t}'', \tilde{\sigma}'', \tilde{i}, \phi \wedge \tilde{i} \sim i$.

□

Appendix E

Assistive TopHat soundness and completeness

E.1 Completeness proofs

E.1.1 Completeness of Simulate

Proof: The structure of the proof of Lemma 7.4.4 is outlined in Fig. 7.4.

For all tasks t and states σ such that $t, \sigma \xRightarrow{I}^* v$, we have by definition of $\xRightarrow{I}^*$ that:

$$t, \sigma \xRightarrow{i_1} t_1, \sigma_1 \xRightarrow{i_2} \dots \xRightarrow{i_n} t_n, \sigma_n \text{ with } \mathcal{V}(t_n, \sigma_n) \text{ and } I = [i_1, \dots, i_n].$$

We need to show that we have $(\tilde{v}, \tilde{I}, \Phi) \in (t, \sigma \approx^* \tilde{v}, \tilde{I}, \Phi)$, which is defined as follows.

$$\begin{aligned} t, \sigma &\approx \tilde{t}_1, \tilde{\sigma}_1, \tilde{i}_1, \phi_1 \\ \tilde{t}_1, \tilde{\sigma}_1 &\approx \tilde{t}_2, \tilde{\sigma}_2, \tilde{i}_2, \phi_2 \\ &\quad \tilde{t}_2, \tilde{\sigma}_2 \approx \dots \\ &\quad \dots \approx \tilde{t}_n, \tilde{\sigma}_n, \tilde{i}_n, \phi_n \end{aligned}$$

with $\mathcal{V}(\tilde{t}_n, \tilde{\sigma}_n) = \tilde{v}$ and $\mathcal{S}(\phi_1 \wedge \dots \wedge \phi_n)$.

By Lemma 7.4.7, we know that $t, \sigma \approx \tilde{t}_1, \tilde{\sigma}_1, \tilde{i}_1, \phi_1$ exists, since $t, \sigma \sqsubseteq_{\emptyset} t, \sigma, \text{True}$. This also gives us that $\tilde{i}_1 \sim i_1$ and $t_1, \sigma_1 \sqsubseteq_{[s_1 \mapsto c_1]} \tilde{t}_1, \tilde{\sigma}_1, \phi_1$ with $\text{SymOf}(\tilde{i}_1) = s_1$ and $\text{ValOf}(i_1) = c_1$.

By repeated application of Lemma 7.4.7, until we arrive at t_n, σ_n , we can show that there exists a $\tilde{I}$ such that $t, \sigma \approx^* \tilde{v}, \tilde{I}, \Phi$, namely $[\tilde{i}_1, \dots, \tilde{i}_n]$. $\square$

E.1.2 Completeness of interaction

Proof: The proof of Lemma 7.4.7 only consists of one case, since the concrete interacting semantics consists of one rule, namely I-HANDLE.

Given that $t, \sigma \sqsubseteq_M \tilde{t}, \tilde{\sigma}, \Phi$ and $t, \sigma \xRightarrow{i} t', \sigma'$, Lemma E.1.1 gives us that there exists a symbolic input $\tilde{i}$ such that $\tilde{i} \sim i$ and $(\tilde{t}', \tilde{\sigma}', \tilde{i}, \phi_1)$ in $\tilde{t}, \tilde{\sigma} \approx \overline{\tilde{t}', \tilde{\sigma}', \tilde{i}, \phi_1}$, with $t', \sigma' \sqsubseteq_{[s \mapsto c]M} \tilde{t}', \tilde{\sigma}', \Phi \wedge \phi_1$ with $s = \text{SymOf}(\tilde{i})$ and $c = \text{ValOf}(i)$.

Then by Lemma E.1.2, given that $t', \sigma' \Downarrow t'', \sigma''$, we obtain that $\mathcal{S}(\Phi \wedge \phi_1)$ implies that $\tilde{t}', \tilde{\sigma}' \Downarrow \overline{\tilde{t}'', \tilde{\sigma}'', \phi_2}$ with $t'', \sigma'' \sqsubseteq_{[s \mapsto c]M} \tilde{t}'', \tilde{\sigma}'', \Phi \wedge \phi_1 \wedge \phi_2$. $\square$

E.1.3 Completeness of handling

Lemma E.1.1 (Completeness of handling)

For all concrete tasks t , concrete states σ , concrete inputs i , symbolic tasks $\tilde{t}$, symbolic states $\tilde{\sigma}$ path conditions Φ and mappings M , we have that $t, \sigma \sqsubseteq_M \tilde{t}, \tilde{\sigma}, \Phi$ and $t, \sigma \xrightarrow{i} t', \sigma'$ together with $\tilde{t}, \tilde{\sigma} \rightsquigarrow \tilde{t}', \tilde{\sigma}', \tilde{i}, \phi$, and for all tuples $(\tilde{t}', \tilde{\sigma}', \tilde{i}, \phi)$ we have that $\mathcal{S}(\Phi \wedge \phi)$ and $\iota \sim i$ implies $t', \sigma' \sqsubseteq_{[s \mapsto c]M} \tilde{t}', \tilde{\sigma}', \Phi \wedge \phi$ where $\text{SymOf}(\tilde{i}) = s$ and $\text{ValOf}(i) = c$.

Proof: We prove Lemma E.1.1 by induction over the derivation $t, \sigma \xrightarrow{i} t', \sigma'$.

Case H-CHANGE

Provided that $\Box v, \sigma \sqsubseteq_M \tilde{t}, \tilde{\sigma}, \Phi$ and $\Box v, \sigma \xrightarrow{v'} \Box v', \sigma$ with $v, v' : \beta$, then by SH-CHANGE $\Box \tilde{v}, \tilde{\sigma} \rightsquigarrow \Box s, \tilde{\sigma}, s, \text{True}$. $\mathcal{S}(\Phi \wedge \text{True}) = \mathcal{S}(\Phi)$, which follows from the premise. Furthermore we have $s \sim v'$ by definition of input simulation. Then finally $\Box v', \sigma \sqsubseteq_{[s \mapsto v']M} \Box s, \tilde{\sigma}, \Phi$ since $[s \mapsto v']Ms = v'$.

Case H-FILL

Provided that $\Box \beta, \sigma \sqsubseteq_M \tilde{t}, \tilde{\sigma}, \Phi$ and $\Box \beta, \sigma \xrightarrow{v} \Box v, \sigma$ with $v : \beta$ then $\Box \beta, \tilde{\sigma} \rightsquigarrow \Box s, \tilde{\sigma}, s, \text{True}$ by SH-FILL. $\mathcal{S}(\Phi \wedge \text{True}) = \mathcal{S}(\Phi)$, which follows from the premise. Furthermore we have $s \sim v$ by definition of input simulation. Then finally $\Box v, \sigma \sqsubseteq_{[s \mapsto v]M} \Box s, \tilde{\sigma}, \Phi$ since $[s \mapsto v]Ms = v$.

Case H-UPDATE

Provided that $\blacksquare l, \sigma \sqsubseteq_M \tilde{t}, \tilde{\sigma}, \Phi$ and $\blacksquare l, \sigma \xrightarrow{v} \blacksquare l, \sigma[l \mapsto v]$ with $\sigma(l), v : \beta$, then $\blacksquare l, \tilde{\sigma} \rightsquigarrow \blacksquare l, \tilde{\sigma}[l \mapsto s], s, \text{True}$ by SH-UPDATE. $\mathcal{S}(\Phi \wedge \text{True}) = \mathcal{S}(\Phi)$, which follows from the premise. Furthermore we have $s \sim v$ by definition of input simulation. Then finally $\blacksquare l, \sigma[l \mapsto v] \sqsubseteq_{[s \mapsto v]M} \blacksquare l, \tilde{\sigma}[l \mapsto s], \Phi$ since $[s \mapsto v]Ms = v$.

Case H-NEXT

Provided that $t_1 \triangleright e_2, \sigma \sqsubseteq_M \tilde{t}, \tilde{\sigma}, \Phi$ and $t_1 \triangleright e_2, \sigma \xrightarrow{c} t_2, \sigma'$, then by SH-NEXT we have $\tilde{t}_1 \triangleright \tilde{e}_2, \tilde{\sigma} \rightsquigarrow \tilde{t}'_1 \triangleright \tilde{e}_2, \tilde{\sigma}_1, \tilde{i}, \phi_1 \cup \tilde{t}_2, \tilde{\sigma}_2, C, \phi_2$. The simulation step results in two sets, from which only the second adheres to the requirement that the symbolic input should simulate the concrete input. For this set, $\tilde{t}_2, \tilde{\sigma}'_2, C, \phi_2$, we have $\mathcal{S}(\Phi \wedge \phi_2)$ implies $t_2, \sigma'_2 \sqsubseteq_M \tilde{t}_2, \tilde{\sigma}'_2, \Phi \wedge \phi_2$, which follows directly from Lemma E.1.2.

Case H-PASSNEXT

Provided that $t_1 \triangleright e_2, \sigma \sqsubseteq_M \tilde{t}, \tilde{\sigma}, \Phi$ and $t_1 \triangleright e_2, \sigma \xrightarrow{i} t'_1 \triangleright e_2, \sigma'$, there are three symbolic rules that apply in this case, namely SH-PASSNEXT, SH-PASSNEXTFAIL and SH-NEXT.

We are only interested in the runs that produce a symbolic input that simulates the concrete input i . Whichever rule applies, we deal with the same premise because of this restriction. This allows us to apply the induction hypothesis and obtain that $\mathcal{S}(\Phi \wedge \phi_1)$ and $\tilde{t} \sim i$ implies $t'_1, \sigma' \sqsubseteq_{[s \mapsto c]M} \tilde{t}'_1, \tilde{\sigma}', \Phi \wedge \phi_1$ with $\text{SymOf}(\tilde{t}) = s$ and $\text{ValOf}(i) = c$. From this, we can directly conclude that $t'_1 \triangleright e_2, \sigma' \sqsubseteq_{[s \mapsto c]M} \tilde{t}'_1 \triangleright \tilde{e}_2, \tilde{\sigma}', \Phi \wedge \phi_1$.

Case H-PASSTHEN

Provided that $t_1 \blacktriangleright e_2, \sigma \sqsubseteq_M \tilde{t}, \tilde{\sigma}, \Phi$ and $t_1 \blacktriangleright e_2, \sigma \xrightarrow{i} t'_1 \blacktriangleright e_2, \sigma'$, then $\tilde{t} \blacktriangleright \tilde{e}_2, \tilde{\sigma} \rightsquigarrow \tilde{t}'_1 \blacktriangleright e_2, \tilde{\sigma}', \tilde{t}, \phi$ by SH-PASSTHEN.

By the induction hypothesis, we obtain $\mathcal{S}(\Phi \wedge \phi)$ and $\tilde{t} \sim i$ implies $t'_1, \sigma' \sqsubseteq_{[s \mapsto c]M} \tilde{t}'_1, \tilde{\sigma}', \Phi \wedge \phi$ with $\text{SymOf}(\tilde{t}) = s$ and $\text{ValOf}(i) = c$ from which we can conclude that $t'_1 \blacktriangleright e_2, \sigma' \sqsubseteq_{[s \mapsto c]M} \tilde{t}'_1 \blacktriangleright \tilde{e}_2, \tilde{\sigma}', \Phi \wedge \phi$.

Case H-PICKLEFT

Provided that $e_1 \diamond e_2, \sigma \sqsubseteq_M \tilde{t}, \tilde{\sigma}, \Phi$ and $e_1 \diamond e_2, \sigma \xrightarrow{L} t_1, \sigma'$, then by SH-PICK we have $\tilde{e}_1 \diamond \tilde{e}_2, \tilde{\sigma} \rightsquigarrow \tilde{t}_1, \tilde{\sigma}_1, L, \phi_1 \cup \tilde{t}_2, \tilde{\sigma}_2, R, \phi_2$.

By Lemma E.1.2 we obtain $\mathcal{S}(\Phi \wedge \phi_1)$ implies $t_1, \sigma' \sqsubseteq_M \tilde{t}_1, \tilde{\sigma}', \Phi \wedge \phi_1$ from which we can conclude that $t_1, \sigma' \sqsubseteq_M \tilde{t}_1, \tilde{\sigma}', \Phi \wedge \phi_1$.

Case H-PICKRIGHT

Provided that $e_1 \diamond e_2, \sigma \sqsubseteq_M \tilde{t}, \tilde{\sigma}, \Phi$ and $e_1 \diamond e_2, \sigma \xrightarrow{R} t_2, \sigma'$, then $\tilde{e}_1 \diamond \tilde{e}_2, \tilde{\sigma} \rightsquigarrow \tilde{t}_1, \tilde{\sigma}_1, L, \phi_1 \cup \tilde{t}_2, \tilde{\sigma}_2, R, \phi_2$ by SH-PICK.

By Lemma E.1.2 we obtain $\mathcal{S}(\Phi \wedge \phi_2)$ implies $t_2, \sigma' \sqsubseteq_M \tilde{t}_2, \tilde{\sigma}', \Phi \wedge \phi_2$ from which we can conclude that $t_2, \sigma' \sqsubseteq_M \tilde{t}_2, \tilde{\sigma}', \Phi \wedge \phi_2$.

Case H-FIRSTOR

Provided that $t_1 \blacklozenge t_2, \sigma \sqsubseteq_M \tilde{t}, \tilde{\sigma}, \Phi$ and $t_1 \blacklozenge t_2, \sigma \xrightarrow{Fi} t'_1 \blacklozenge t_2, \sigma'$, then $\tilde{t}_1 \blacklozenge \tilde{t}_2, \tilde{\sigma} \rightsquigarrow \tilde{t}_1, \tilde{\sigma}_1, F\tilde{t}, \phi_1 \cup \tilde{t}_2, \tilde{\sigma}_2, S\tilde{t}, \phi_2$ by SH-OR.

By the induction hypothesis we obtain $\mathcal{S}(\Phi \wedge \phi_1)$ implies $t'_1, \sigma' \sqsubseteq_M \tilde{t}'_1, \tilde{\sigma}', \Phi \wedge \phi_1$ from which we can conclude $t'_1 \blacklozenge t_2, \sigma' \sqsubseteq_M \tilde{t}'_1 \blacklozenge \tilde{t}_2, \tilde{\sigma}', \Phi \wedge \phi_1$.

Case H-SECONDOOR

Provided that $t_1 \blacklozenge t_2, \sigma \sqsubseteq_M \tilde{t}, \tilde{\sigma}, \Phi$ and $t_1 \blacklozenge t_2, \sigma \xrightarrow{Si} t_1 \blacklozenge t'_2, \sigma'$, then $\tilde{t}_1 \blacklozenge \tilde{t}_2, \tilde{\sigma} \rightsquigarrow \tilde{t}_1, \tilde{\sigma}_1, F\tilde{t}, \phi_1 \cup \tilde{t}_2, \tilde{\sigma}_2, S\tilde{t}, \phi_2$ by SH-OR.

By application of the induction hypothesis we obtain $\mathcal{S}(\Phi \wedge \phi_2)$ implies $t'_2, \sigma' \sqsubseteq_M \tilde{t}'_2, \tilde{\sigma}', \Phi \wedge \phi_2$ from which we can conclude that $t_1 \blacklozenge t'_2, \sigma' \sqsubseteq_M \tilde{t}_1 \blacklozenge \tilde{t}'_2, \tilde{\sigma}', \Phi \wedge \phi_2$.

Case H-FIRSTAND

Provided that $t_1 \bowtie t_2, \sigma \sqsubseteq_M \tilde{t}, \tilde{\sigma}, \Phi$ and $t_1 \bowtie t_2, \sigma \xrightarrow{Fi} t'_1 \bowtie t_2, \sigma'$, then $\tilde{t}_1 \bowtie \tilde{t}_2, \tilde{\sigma} \rightsquigarrow \tilde{t}'_1 \bowtie \tilde{t}_2, \tilde{\sigma}_1, F \tilde{t}_1, \phi_1 \cup \tilde{t}_1 \bowtie \tilde{t}'_2, \tilde{\sigma}_2, S \tilde{t}_2, \phi_2$ by SH-AND.

By application of the induction hypothesis we obtain $\mathcal{S}(\Phi \wedge \phi_1)$ implies $t'_1, \sigma' \sqsubseteq_M \tilde{t}'_1, \tilde{\sigma}', \Phi \wedge \phi_1$ from which we can conclude that $t'_1 \bowtie t_2, \sigma' \sqsubseteq_M \tilde{t}'_1 \bowtie \tilde{t}_2, \tilde{\sigma}', \Phi \wedge \phi_1$.

Case H-SECONDAND

Provided that $t_1 \bowtie t_2, \sigma \sqsubseteq_M \tilde{t}, \tilde{\sigma}, \Phi$ and $t_1 \bowtie t_2, \sigma \xrightarrow{Si} t_1 \bowtie t'_2, \sigma'$, then $\tilde{t}_1 \bowtie \tilde{t}_2, \tilde{\sigma} \rightsquigarrow \tilde{t}'_1 \bowtie \tilde{t}_2, \tilde{\sigma}_1, F \tilde{t}_1, \phi_1 \cup \tilde{t}_1 \bowtie \tilde{t}'_2, \tilde{\sigma}_2, S \tilde{t}_2, \phi_2$ by SH-AND.

By application of the induction hypothesis we obtain $\mathcal{S}(\Phi \wedge \phi_2)$ implies $t'_2, \sigma' \sqsubseteq_M \tilde{t}'_2, \tilde{\sigma}', \Phi \wedge \phi_2$ from which we can conclude that $t_1 \bowtie t'_2, \sigma' \sqsubseteq_M \tilde{t}_1 \bowtie \tilde{t}'_2, \tilde{\sigma}', \Phi \wedge \phi_2$.

□

E.1.4 Completeness of normalisation

Lemma E.1.2 (Completeness of normalisation)

For all concrete expressions e , concrete states σ , symbolic expressions $\tilde{e}$, symbolic states $\tilde{\sigma}$ path conditions Φ and mappings M , if $e, \sigma \sqsubseteq_M \tilde{e}, \tilde{\sigma}, \Phi$ and $e, \sigma \Downarrow t, \sigma'$ then $\tilde{e}, \tilde{\sigma} \Downarrow \tilde{t}, \tilde{\sigma}', \phi$ and for all tuples $(\tilde{t}, \tilde{\sigma}', \phi)$ we have that $\mathcal{S}(\Phi \wedge \phi)$ implies $t, \sigma' \sqsubseteq_M \tilde{t}, \tilde{\sigma}', \Phi \wedge \phi$.

Proof: We prove Lemma E.1.2 by induction over the derivation $e, \sigma \Downarrow t, \sigma'$.

The base case is when the N-DONE rule applies.

In this case, we obtain from Lemma E.1.4 that $\tilde{e}, \tilde{\sigma} \Downarrow \tilde{t}, \tilde{\sigma}', \phi$ with $t, \sigma' \sqsubseteq_M \tilde{t}, \tilde{\sigma}', \Phi \wedge \phi$. Together with SN-DONE this gives us $\tilde{e}, \tilde{\sigma} \Downarrow \tilde{t}, \tilde{\sigma}'$, which is exactly what we need to show.

The only induction step is when N-REPEAT applies. In this case, we obtain from Lemma E.1.4 that $\tilde{e}, \tilde{\sigma} \Downarrow \tilde{t}, \tilde{\sigma}', \phi_1$ with $t, \sigma' \sqsubseteq_M \tilde{t}, \tilde{\sigma}', \Phi \wedge \phi_1$. Furthermore, by Lemma E.1.3 we obtain that $\tilde{t}, \tilde{\sigma}' \rightsquigarrow \tilde{t}', \tilde{\sigma}'', \phi_2$ with $t', \sigma'' \sqsubseteq_M \tilde{t}', \tilde{\sigma}'', \Phi \wedge \phi_1 \wedge \phi_2$. Then finally, by application of the induction hypothesis we obtain $\tilde{t}', \tilde{\sigma}'' \Downarrow \tilde{t}'', \tilde{\sigma}''', \phi_3$ with $t'', \sigma''' \sqsubseteq_M \tilde{t}'', \tilde{\sigma}''', \Phi \wedge \phi_1 \wedge \phi_2 \wedge \phi_3$. The above, together with the SN-REPEAT rule gives us that: $\tilde{e}, \tilde{\sigma} \Downarrow \tilde{t}'', \tilde{\sigma}''', \phi_1 \wedge \phi_2 \wedge \phi_3$ with $t'', \sigma''' \sqsubseteq_M \tilde{t}'', \tilde{\sigma}''', \Phi \wedge \phi_1 \wedge \phi_2 \wedge \phi_3$. □

E.1.5 Completeness of striding

Lemma E.1.3 (Completeness of striding)

For all concrete tasks t , concrete states σ , symbolic tasks $\tilde{t}$, symbolic states $\tilde{\sigma}$ path conditions Φ and mappings M , we have that if $t, \sigma \sqsubseteq_M \tilde{t}, \tilde{\sigma}, \Phi$ and $t, \sigma \mapsto t', \sigma'$, then $\tilde{t}, \tilde{\sigma} \rightsquigarrow \tilde{t}', \tilde{\sigma}', \phi$, and for all tuples $(\tilde{t}', \tilde{\sigma}', \phi)$ we have that $S(\Phi \wedge \phi)$ implies $t', \sigma' \sqsubseteq_M \tilde{t}', \tilde{\sigma}', \Phi \wedge \phi$.

Proof: We prove Lemma E.1.3 by induction over the derivation $t, \sigma \mapsto t', \sigma'$.

Case S-EDIT

Provided that $\Box v, \sigma \sqsubseteq_M \tilde{t}, \tilde{\sigma}, \Phi$ and $\Box v, \sigma \mapsto \Box v, \sigma$, we can conclude that $\tilde{t} = \Box \tilde{v}$ and then by SS-EDIT, $\Box \tilde{v}, \tilde{\sigma} \rightsquigarrow \Box \tilde{v}, \tilde{\sigma}$. Since the expressions do not change in this case, consistency holds trivially.

Case S-FILL

Provided that $\Box \beta, \sigma \sqsubseteq_M \tilde{t}, \tilde{\sigma}, \Phi$ and $\Box \beta, \sigma \mapsto \Box \beta, \sigma$, we can conclude that $\tilde{t} = \Box \beta$ and then by SS-FILL, $\Box \beta, \tilde{\sigma} \rightsquigarrow \Box \beta, \tilde{\sigma}$. Since the expressions do not change in this case, consistency holds trivially.

Case S-UPDATE

Provided that $\blacksquare l, \sigma \sqsubseteq_M \tilde{t}, \tilde{\sigma}, \Phi$ and $\blacksquare l, \sigma \mapsto \blacksquare l, \sigma$, we can conclude that $\tilde{t} = \blacksquare l$ and then by SS-UPDATE, $\blacksquare l, \tilde{\sigma} \rightsquigarrow \blacksquare l, \tilde{\sigma}$. Since the expressions do not change in this case, consistency holds trivially.

Case S-FAIL

Provided that $\downarrow, \sigma \sqsubseteq_M \tilde{t}, \tilde{\sigma}, \Phi$ and $\downarrow, \sigma \mapsto \downarrow, \sigma$, we can conclude that $\tilde{t} = \downarrow$ and then by SS-FAIL, $\downarrow, \tilde{\sigma} \rightsquigarrow \downarrow, \tilde{\sigma}$. Since the expressions do not change in this case, consistency holds trivially.

Case S-XOR

Provided that $e_1 \Diamond e_2, \sigma \sqsubseteq_M \tilde{t}, \tilde{\sigma}, \Phi$ and $e_1 \Diamond e_2, \sigma \mapsto e_1 \Diamond e_2, \sigma$, we can conclude that $\tilde{t} = \tilde{e}_1 \Diamond \tilde{e}_2$ and then by SS-XOR, $\tilde{e}_1 \Diamond \tilde{e}_2, \tilde{\sigma} \rightsquigarrow \tilde{e}_1 \Diamond \tilde{e}_2, \tilde{\sigma}$. Since the expressions do not change in this case, consistency holds trivially.

Case S-THENSTAY, S-THENFAIL

Provided that $t_1 \blacktriangleright e_2, \sigma \sqsubseteq_M \tilde{t}, \tilde{\sigma}, \Phi$ and $t_1 \blacktriangleright e_2, \sigma \mapsto t'_1 \blacktriangleright e_2, \sigma'$, then by the induction hypothesis, we have $\tilde{t}_1, \tilde{\sigma} \rightsquigarrow \tilde{t}'_1, \tilde{\sigma}', \phi$ and $t'_1, \sigma' \sqsubseteq_M \tilde{t}'_1, \tilde{\sigma}', \Phi \wedge \phi$. Then by SS-THENSTAY and SS-THENFAIL respectively, we have $\tilde{t}_1 \blacktriangleright \tilde{e}_2, \sigma \rightsquigarrow \tilde{t}'_1 \blacktriangleright \tilde{e}_2, \sigma', \phi$ and $t'_1 \blacktriangleright e_2, \sigma' \sqsubseteq_M \tilde{t}'_1 \blacktriangleright \tilde{e}_2, \tilde{\sigma}', \Phi \wedge \phi$.

Case S-THENCONT

Provided that $t_1 \blacktriangleright e_2, \sigma \sqsubseteq_M \tilde{t}, \tilde{\sigma}, \Phi$ and $t_1 \blacktriangleright e_2, \sigma \mapsto t_2, \sigma''$, then by the induction hypothesis, we have $\tilde{t}_1, \tilde{\sigma} \rightsquigarrow \tilde{t}'_1, \tilde{\sigma}', \phi$ and $t'_1, \sigma' \sqsubseteq_M \tilde{t}'_1, \tilde{\sigma}', \Phi \wedge \phi_1$.

From the premise, we have $\mathcal{V}(t'_1, \sigma') = v_1$. This gives us that we also have $\mathcal{V}(\tilde{t}'_1, \tilde{\sigma}') = \tilde{v}_1$. Lemma E.1.4 gives us that $\tilde{e}_2 \tilde{v}_1, \tilde{\sigma}' \Downarrow \tilde{t}_2, \tilde{\sigma}'', \phi_2$ and $t_2, \sigma'' \sqsubseteq_M \tilde{t}_2, \tilde{\sigma}'', \Phi \wedge \phi_1 \wedge \phi_2$.

Then by SS-THENCONT, we have $\tilde{t}_1 \blacktriangleright \tilde{e}_2, \sigma \rightsquigarrow \tilde{t}_2, \sigma'', \phi_1 \wedge \phi_2$ and $t_2, \sigma'' \sqsubseteq_M \tilde{t}_2, \tilde{\sigma}'', \Phi \wedge \phi_1 \wedge \phi_2$.

Case S-ORLEFT

Provided that $t_1 \blacklozenge t_2, \sigma \sqsubseteq_M \tilde{t}, \tilde{\sigma}, \Phi$ and $t_1 \blacklozenge t_2, \sigma \mapsto t'_1, \sigma'$, then by the induction hypothesis, we have $\tilde{t}_1, \tilde{\sigma} \rightsquigarrow \tilde{t}'_1, \tilde{\sigma}', \phi$ and $t'_1, \sigma' \sqsubseteq_M \tilde{t}'_1, \tilde{\sigma}', \Phi \wedge \phi$. Then by SS-ORLEFT, we have $\tilde{t}_1 \blacklozenge \tilde{t}_2, \sigma \rightsquigarrow \tilde{t}'_1, \sigma', \phi$ and $t'_1, \sigma' \sqsubseteq_M \tilde{t}'_1, \tilde{\sigma}', \Phi \wedge \phi$.

Case S-ORRIGHT

Provided that $t_1 \blacklozenge t_2, \sigma \sqsubseteq_M \tilde{t}, \tilde{\sigma}, \Phi$ and $t_1 \blacklozenge t_2, \sigma \mapsto t'_2, \sigma''$, then by the induction hypothesis, we have $\tilde{t}_1, \tilde{\sigma} \rightsquigarrow \tilde{t}'_1, \tilde{\sigma}', \phi$ and $t'_1, \sigma' \sqsubseteq_M \tilde{t}'_1, \tilde{\sigma}', \Phi \wedge \phi_1$. A second application of the induction hypothesis gives us that $\tilde{t}_2, \tilde{\sigma}' \rightsquigarrow \tilde{t}'_2, \tilde{\sigma}'', \phi_2$ and $t'_2, \sigma'' \sqsubseteq_M \tilde{t}'_2, \tilde{\sigma}'', \Phi \wedge \phi_1 \wedge \phi_2$. Then by SS-ORRIGHT, we have $\tilde{t}_1 \blacklozenge \tilde{t}_2, \sigma \rightsquigarrow \tilde{t}'_1, \sigma'', \phi_1 \wedge \phi_2$ and $t'_2, \sigma'' \sqsubseteq_M \tilde{t}'_2, \tilde{\sigma}'', \Phi \wedge \phi_1 \wedge \phi_2$.

Case S-ORNONE

Provided that $t_1 \blacklozenge t_2, \sigma \sqsubseteq_M \tilde{t}, \tilde{\sigma}, \Phi$ and $t_1 \blacklozenge t_2, \sigma \mapsto t'_1 \blacklozenge t'_2, \sigma''$, then by the induction hypothesis, we have $\tilde{t}_1, \tilde{\sigma} \rightsquigarrow \tilde{t}'_1, \tilde{\sigma}', \phi$ and $t'_1, \sigma' \sqsubseteq_M \tilde{t}'_1, \tilde{\sigma}', \Phi \wedge \phi_1$. A second application of the induction hypothesis gives us that $\tilde{t}_2, \tilde{\sigma}' \rightsquigarrow \tilde{t}'_2, \tilde{\sigma}'', \phi_2$ and $t'_2, \sigma'' \sqsubseteq_M \tilde{t}'_2, \tilde{\sigma}'', \Phi \wedge \phi_1 \wedge \phi_2$. Then by SS-ORNONE, we have $\tilde{t}_1 \blacklozenge \tilde{t}_2, \sigma \rightsquigarrow \tilde{t}'_1 \blacklozenge \tilde{t}'_2, \sigma'', \phi_1 \wedge \phi_2$ and $t'_1 \blacklozenge t'_2, \sigma'' \sqsubseteq_M \tilde{t}'_1 \blacklozenge \tilde{t}'_2, \sigma'', \Phi \wedge \phi_1 \wedge \phi_2$.

Case S-NEXT

Provided that $t_1 \triangleright e_2, \sigma \sqsubseteq_M \tilde{t}, \tilde{\sigma}, \Phi$ and $t_1 \triangleright e_2, \sigma \mapsto t'_1 \triangleright e_2, \sigma'$, then by the induction hypothesis, we have $\tilde{t}_1, \tilde{\sigma} \rightsquigarrow \tilde{t}'_1, \tilde{\sigma}', \phi$ and $t'_1, \sigma' \sqsubseteq_M \tilde{t}'_1, \tilde{\sigma}', \Phi \wedge \phi$. Then by SS-NEXT, we have $\tilde{t}_1 \triangleright \tilde{e}_2, \sigma \rightsquigarrow \tilde{t}'_1 \triangleright \tilde{e}_2, \sigma', \phi$ and $t'_1 \triangleright e_2, \sigma' \sqsubseteq_M \tilde{t}'_1 \triangleright \tilde{e}_2, \sigma', \Phi \wedge \phi$.

Case S-AND

This case is proven in the same way as S-ORNONE.

□

E.1.6 Completeness of evaluate

Lemma E.1.4 (Completeness of evaluate)

For all concrete expressions e , concrete states σ , symbolic expressions $\tilde{e}$, symbolic states $\tilde{\sigma}$ path conditions Φ and mappings M , we have that if $e, \sigma \sqsubseteq_M \tilde{e}, \tilde{\sigma}, \Phi$ and $e, \sigma \downarrow v, \sigma'$, then $\tilde{e}, \tilde{\sigma} \downarrow \tilde{v}, \tilde{\sigma}', \phi$, and for all tuples $(\tilde{v}, \tilde{\sigma}', \phi)$ we have that $\mathcal{S}(\Phi \wedge \phi)$ implies $v, \sigma' \sqsubseteq_M \tilde{v}, \tilde{\sigma}', \Phi \wedge \phi$.

Proof: We prove Lemma E.1.4 by induction over the derivation $e, \sigma \downarrow v, \sigma'$.

Case E-VALUE

Provided that $v, \sigma \sqsubseteq_M \tilde{e}, \tilde{\sigma}, \Phi$ and $v, \sigma \downarrow v, \sigma$, we know that $\tilde{e} = \tilde{v}$. By SE-VALUE, we have $\tilde{v}, \tilde{\sigma} \downarrow \tilde{v}, \tilde{\sigma}, \text{True}$. Since the expressions did not change, this case holds trivially.

Case E-PAIR

Provided that $\langle e_1, e_2 \rangle, \sigma \sqsubseteq_M \tilde{e}, \tilde{\sigma}, \Phi$ and $\langle e_1, e_2 \rangle, \sigma \downarrow \langle v_1, v_2 \rangle, \sigma''$, then by application of the induction hypothesis we obtain $\tilde{e}_1, \tilde{\sigma} \downarrow \tilde{v}_1, \tilde{\sigma}', \phi_1$ and $v_1, \sigma' \sqsubseteq_M \tilde{v}_1, \tilde{\sigma}', \Phi \wedge \phi_1$. A second application of the induction hypothesis gives us $\tilde{e}_2, \tilde{\sigma}' \downarrow \tilde{v}_2, \tilde{\sigma}'', \phi_2$ and $v_2, \sigma'' \sqsubseteq_M \tilde{v}_2, \tilde{\sigma}'', \Phi \wedge \phi_2$. By SE-PAIR, we have $\langle \tilde{e}_1, \tilde{e}_2 \rangle, \tilde{\sigma} \downarrow \langle \tilde{v}_1, \tilde{v}_2 \rangle, \tilde{\sigma}'', \phi_1 \wedge \phi_2$ and $\langle v_1, v_2 \rangle, \sigma'' \sqsubseteq_M \langle \tilde{v}_1, \tilde{v}_2 \rangle, \tilde{\sigma}'', \Phi \wedge \phi_1 \wedge \phi_2$.

Case E-FIRST

Provided that $\text{fst } e, \sigma \sqsubseteq_M \tilde{e}, \tilde{\sigma}, \Phi$ and $\text{fst } e, \sigma \downarrow v_1, \sigma'$, then by application of the induction hypothesis we obtain $\tilde{e}, \tilde{\sigma} \downarrow \langle \tilde{v}_1, \tilde{v}_2 \rangle, \tilde{\sigma}', \phi$ and $\langle v_1, v_2 \rangle, \sigma' \sqsubseteq_M \langle \tilde{v}_1, \tilde{v}_1 \rangle, \tilde{\sigma}', \Phi \wedge \phi$. By SE-FIRST, we have $\text{fst } \tilde{e}, \tilde{\sigma} \downarrow \tilde{v}_1, \tilde{\sigma}', \phi$.

Case E-SECOND

Provided that $\text{snd } e, \sigma \sqsubseteq_M \tilde{e}, \tilde{\sigma}, \Phi$ and $\text{snd } e, \sigma \downarrow v_2, \sigma'$, then by application of the induction hypothesis we obtain $\tilde{e}, \tilde{\sigma} \downarrow \langle \tilde{v}_1, \tilde{v}_2 \rangle, \tilde{\sigma}', \phi$ and $\langle v_1, v_2 \rangle, \sigma' \sqsubseteq_M \langle \tilde{v}_1, \tilde{v}_2 \rangle, \tilde{\sigma}', \Phi \wedge \phi$.

By SE-SECOND, we have $\text{snd } \tilde{e}, \tilde{\sigma} \downarrow \tilde{v}_2, \tilde{\sigma}', \phi$.

Case E-CONS

Provided that $e_1 :: e_2, \sigma \sqsubseteq_M \tilde{e}, \tilde{\sigma}, \Phi$ and $e_1 :: e_2, \sigma \downarrow v_1 :: v_2, \sigma''$ then by application of the induction hypothesis we obtain $\tilde{e}_1, \tilde{\sigma} \downarrow \tilde{v}_1, \tilde{\sigma}', \phi_1$ and $v_1, \sigma' \sqsubseteq_M \tilde{v}_1, \tilde{\sigma}', \Phi \wedge \phi_1$. A second application of the induction hypothesis gives us $\tilde{e}_2, \tilde{\sigma}' \downarrow \tilde{v}_2, \tilde{\sigma}'', \phi_2$ and $v_2, \sigma'' \sqsubseteq_M \tilde{v}_2, \tilde{\sigma}'', \Phi \wedge \phi_2$.

By SE-CONS, we have $\tilde{e}_1 :: \tilde{e}_2, \tilde{\sigma} \Downarrow \tilde{v}_1 :: \tilde{v}_2, \tilde{\sigma}'', \phi_1 \wedge \phi_2$ and $v_1 :: v_2, \sigma'' \sqsubseteq_M \tilde{v}_1 :: \tilde{v}_2, \tilde{\sigma}'', \Phi \wedge \phi_1 \wedge \phi_2$.

Case E-HEAD

Provided that $\text{head } e, \sigma \sqsubseteq_M \tilde{e}, \tilde{\sigma}, \Phi$ and $\text{head } e, \sigma \Downarrow v_1, \sigma'$, then by application of the induction hypothesis we obtain $\tilde{e}, \tilde{\sigma} \Downarrow \tilde{v}_1 :: \tilde{v}_2, \tilde{\sigma}', \phi$ and $v_1 :: v_2, \sigma' \sqsubseteq_M \tilde{v}_1 :: \tilde{v}_2, \tilde{\sigma}', \Phi \wedge \phi$. By SE-HEAD, we have $\text{head } \tilde{v}_1 :: \tilde{v}_2, \tilde{\sigma} \Downarrow \tilde{v}_1, \tilde{\sigma}', \phi$.

Case E-TAIL

Provided that $\text{tail } e, \sigma \sqsubseteq_M \tilde{e}, \tilde{\sigma}, \Phi$ and $\text{tail } e, \sigma \Downarrow v_2, \sigma'$, then by application of the induction hypothesis we obtain $\tilde{e}, \tilde{\sigma} \Downarrow \tilde{v}_1 :: \tilde{v}_2, \tilde{\sigma}', \phi$ and $v_1 :: v_2, \sigma' \sqsubseteq_M \tilde{v}_1 :: \tilde{v}_2, \tilde{\sigma}', \Phi \wedge \phi$. By SE-TAIL, we have $\text{tail } \tilde{v}_1 :: \tilde{v}_2, \tilde{\sigma} \Downarrow \tilde{v}_2, \tilde{\sigma}', \phi$.

Case E-APP

Provided that $e_1 e_2, \sigma \sqsubseteq_M \tilde{e}, \tilde{\sigma}, \Phi$ and $e_1 e_2, \sigma \Downarrow v_1, \sigma'''$, then by application of the induction hypothesis we obtain $\tilde{e}_1, \tilde{\sigma} \Downarrow \lambda x : \tau. \tilde{e}'_1, \tilde{\sigma}', \phi_1$ and $\lambda x : \tau. \tilde{e}'_1, \sigma' \sqsubseteq_M \lambda x : \tau. \tilde{e}'_1, \tilde{\sigma}', \Phi \wedge \phi_1$. A second application of the induction hypothesis gives us $\tilde{e}_2, \tilde{\sigma}' \Downarrow \tilde{v}_2, \tilde{\sigma}'', \phi_2$ and $v_2, \sigma'' \sqsubseteq_M \tilde{v}_2, \tilde{\sigma}'', \Phi \wedge \phi_1 \wedge \phi_2$.

Then finally by a third application of the induction hypothesis, we get $\tilde{e}'_1[x \mapsto \tilde{v}_2], \tilde{\sigma}'' \Downarrow \tilde{v}_1, \tilde{\sigma}''', \phi_3$ and $v_1, \sigma''' \sqsubseteq_M \tilde{v}_1, \tilde{\sigma}''', \Phi \wedge \phi_1 \wedge \phi_2 \wedge \phi_3$.

By SE-APP, we have $\tilde{e}_1 \tilde{e}_2, \tilde{\sigma} \Downarrow \tilde{v}_1, \tilde{\sigma}''', \phi_1 \wedge \phi_2 \wedge \phi_2$.

Case E-IFTRUE

Provided that **if** e_1 **then** e_2 **else** $e_3, \sigma \sqsubseteq_M \tilde{e}, \tilde{\sigma}, \Phi$ and **if** e_1 **then** e_2 **else** $e_3, \sigma \Downarrow v_2, \sigma''$, then by application of the induction hypothesis we obtain $\tilde{e}_1, \tilde{\sigma} \Downarrow \tilde{v}_1, \tilde{\sigma}', \phi_1$ and $\text{True}, \sigma' \sqsubseteq_M \tilde{v}_1, \tilde{\sigma}', \Phi \wedge \phi_1$. A second application of the induction hypothesis gives us $\tilde{e}_2, \tilde{\sigma}' \Downarrow \tilde{v}_2, \tilde{\sigma}'', \phi_2$ and $v_2, \sigma'' \sqsubseteq_M \tilde{v}_2, \tilde{\sigma}'', \Phi \wedge \phi_1 \wedge \phi_2$.

By SE-IF, we have **if** $\tilde{e}_1$ **then** $\tilde{e}_2$ **else** $\tilde{e}_3, \tilde{\sigma} \Downarrow \tilde{v}_2, \tilde{\sigma}'', \phi_1 \wedge \phi_2 \wedge \tilde{v}_1$.

Case E-IFFALSE

Provided that **if** e_1 **then** e_2 **else** $e_3, \sigma \sqsubseteq_M \tilde{e}, \tilde{\sigma}, \Phi$ and **if** e_1 **then** e_2 **else** $e_3, \sigma \Downarrow v_3, \sigma''$, then by application of the induction hypothesis we obtain $\tilde{e}_1, \tilde{\sigma} \Downarrow \tilde{v}_1, \tilde{\sigma}', \phi_1$ and $\text{False}, \sigma' \sqsubseteq_M \tilde{v}_1, \tilde{\sigma}', \Phi \wedge \phi_1$. A second application of the induction hypothesis gives us $\tilde{e}_3, \tilde{\sigma}' \Downarrow \tilde{v}_3, \tilde{\sigma}'', \phi_2$ and $v_3, \sigma'' \sqsubseteq_M \tilde{v}_3, \tilde{\sigma}'', \Phi \wedge \phi_1 \wedge \phi_2$.

By SE-IF, we have **if** $\tilde{e}_1$ **then** $\tilde{e}_2$ **else** $\tilde{e}_3, \tilde{\sigma} \Downarrow \tilde{v}_3, \tilde{\sigma}'', \phi_1 \wedge \phi_3 \wedge \neg \tilde{v}_1$.

Case E-REF

Provided that $\text{ref } e, \sigma \sqsubseteq_M \tilde{e}, \tilde{\sigma}, \Phi$ and $\text{ref } e, \sigma \downarrow l, \sigma'[l \mapsto v]$, then by application of the induction hypothesis we obtain $\tilde{e}, \tilde{\sigma} \zeta \tilde{v}, \tilde{\sigma}', \phi$ and $v, \sigma' \sqsubseteq_M \tilde{v}, \tilde{\sigma}', \Phi \wedge \phi$. By SE-REF, we have $\text{ref } \tilde{e}, \tilde{\sigma} \zeta l, \tilde{\sigma}'[l \mapsto \tilde{v}], \phi$ and $l, \sigma'[l \mapsto v] \sqsubseteq_M l, \tilde{\sigma}'[l \mapsto \tilde{v}], \Phi \wedge \phi$.

Case E-DEREF

Provided that $!e, \sigma \sqsubseteq_M \tilde{e}, \tilde{\sigma}, \Phi$ and $!e, \sigma \downarrow \sigma'(l), \sigma'$, then by application of the induction hypothesis we obtain $\tilde{e}, \tilde{\sigma} \zeta l, \tilde{\sigma}', \phi$ and $l, \sigma' \sqsubseteq_M l, \tilde{\sigma}', \Phi \wedge \phi$. By SE-DEREF, we have $!\tilde{e}, \tilde{\sigma} \zeta \tilde{\sigma}'(l), \tilde{\sigma}', \phi$ and $\sigma'(l), \sigma' \sqsubseteq_M \tilde{\sigma}'(l), \tilde{\sigma}', \Phi \wedge \phi$.

Case E-ASSIGN

Provided that $e_1 := e_2, \sigma \sqsubseteq_M \tilde{e}, \tilde{\sigma}, \Phi$ and $e_1 := e_2, \sigma \downarrow \langle \rangle, \sigma''[l \mapsto v_2]$ with $e_1, \sigma \downarrow l, \sigma'$ and $e_2, \sigma' \downarrow v_2, \sigma''$, then by application of the induction hypothesis we obtain $\tilde{e}_1, \tilde{\sigma} \zeta l, \tilde{\sigma}', \phi_1$ and $l, \sigma' \sqsubseteq_M l, \tilde{\sigma}', \Phi \wedge \phi_1$. A second application of the induction hypothesis gives us $\tilde{e}_2, \tilde{\sigma}' \zeta \tilde{v}_2, \tilde{\sigma}'', \phi_2$ and $v_2, \sigma'' \sqsubseteq_M \tilde{v}_2, \tilde{\sigma}'', \Phi \wedge \phi_2$. By SE-ASSIGN, we have $\tilde{e}_1 := \tilde{e}_2, \tilde{\sigma} \zeta \langle \rangle, \tilde{\sigma}''[l \mapsto \tilde{v}_2], \phi_1 \wedge \phi_2$ and $\langle \rangle, \sigma''[l \mapsto v_2] \sqsubseteq_M \langle \rangle, \tilde{\sigma}''[l \mapsto \tilde{v}_2], \Phi \wedge \phi_1 \wedge \phi_2$.

Case E-EDIT

Provided that $\Box e, \sigma \sqsubseteq_M \tilde{e}, \tilde{\sigma}, \Phi$ and $\Box e, \sigma \downarrow \Box v, \sigma'$, then by application of the induction hypothesis we obtain $\tilde{e}, \tilde{\sigma} \zeta \tilde{v}, \tilde{\sigma}', \phi$ and $v, \sigma' \sqsubseteq_M \tilde{v}, \tilde{\sigma}', \Phi \wedge \phi$.

By SE-EDIT, we have $\Box \tilde{e}, \tilde{\sigma} \zeta \Box \tilde{v}, \tilde{\sigma}', \phi$ and $\Box v, \sigma' \sqsubseteq_M \Box \tilde{v}, \tilde{\sigma}', \Phi \wedge \phi$.

Case E-UPDATE

Provided that $\blacksquare e, \sigma \sqsubseteq_M \tilde{e}, \tilde{\sigma}, \Phi$ and $\blacksquare e, \sigma \downarrow \blacksquare l, \sigma'$, then by application of the induction hypothesis we obtain $\tilde{e}, \tilde{\sigma} \zeta l, \tilde{\sigma}', \phi$ and $l, \sigma' \sqsubseteq_M l, \tilde{\sigma}', \Phi \wedge \phi$. By SE-UPDATE, we have $\blacksquare \tilde{e}, \tilde{\sigma} \zeta \blacksquare l, \tilde{\sigma}', \phi$ and $\blacksquare l, \sigma' \sqsubseteq_M \blacksquare l, \tilde{\sigma}', \Phi \wedge \phi$.

Case E-THEN

Provided that $e_1 \blacktriangleright e_2, \sigma \sqsubseteq_M \tilde{e}, \tilde{\sigma}, \Phi$ and $e_1 \blacktriangleright e_2, \sigma \downarrow t_1 \blacktriangleright e_2, \sigma'$, then by application of the induction hypothesis we obtain $\tilde{e}_1, \tilde{\sigma} \zeta \tilde{v}_1, \tilde{\sigma}', \phi$ and $v_1, \sigma' \sqsubseteq_M \tilde{v}_1, \tilde{\sigma}', \Phi \wedge \phi$. By SE-THEN, we have $\tilde{e}_1 \blacktriangleright \tilde{e}_2, \tilde{\sigma} \zeta \tilde{v}_1 \blacktriangleright \tilde{e}_2, \tilde{\sigma}', \phi$ and $v_1 \blacktriangleright e_2, \sigma' \sqsubseteq_M \tilde{v}_1 \blacktriangleright \tilde{e}_2, \tilde{\sigma}', \Phi \wedge \phi$.

Case E-NEXT

Provided that $e_1 \triangleright e_2, \sigma \sqsubseteq_M \tilde{e}, \tilde{\sigma}, \Phi$ and $e_1 \triangleright e_2, \sigma \downarrow t_1 \triangleright e_2, \sigma'$, then by application of the induction hypothesis we obtain $\tilde{e}_1, \tilde{\sigma} \zeta \tilde{v}_1, \tilde{\sigma}', \phi$ and

$v_1, \sigma' \sqsubseteq_M \tilde{v}_1, \tilde{\sigma}', \Phi \wedge \phi$. By SE-NEXT, we have $\tilde{e}_1 \triangleright \tilde{e}_2, \tilde{\sigma} \searrow \tilde{v}_1 \triangleright \tilde{e}_2, \tilde{\sigma}', \phi$ and $v_1 \triangleright e_2, \sigma' \sqsubseteq_M \tilde{v}_1 \triangleright \tilde{e}_2, \tilde{\sigma}', \Phi \wedge \phi$.

Case E-OR

Provided that $e_1 \blacklozenge e_2, \sigma \sqsubseteq_M \tilde{e}, \tilde{\sigma}, \Phi$ and $e_1 \blacklozenge e_2, \sigma \downarrow t_1 \blacklozenge t_2, \sigma''$, then by application of the induction hypothesis we obtain $\tilde{e}_1, \tilde{\sigma} \searrow \tilde{t}_1, \tilde{\sigma}', \phi_1$ and $t_1, \sigma' \sqsubseteq_M \tilde{t}_1, \tilde{\sigma}', \Phi \wedge \phi_1$.

A second application of the induction hypothesis gives us

$\tilde{e}_2, \tilde{\sigma}' \searrow \tilde{t}_2, \tilde{\sigma}'', \phi_2$ and $t_2, \sigma'' \sqsubseteq_M \tilde{t}_2, \tilde{\sigma}'', \Phi \wedge \phi_2$.

By SE-OR, we have $\tilde{e}_1 \blacklozenge \tilde{e}_2, \tilde{\sigma} \searrow \tilde{t}_1 \blacklozenge \tilde{t}_2, \tilde{\sigma}'', \phi_1 \wedge \phi_2$ and $t_1 \blacklozenge t_2, \sigma'' \sqsubseteq_M \tilde{t}_1 \blacklozenge \tilde{t}_2, \tilde{\sigma}'', \Phi \wedge \phi_1 \wedge \phi_2$.

□

E.2 Soundness proofs

E.2.1 Soundness of simulate

Proof: The structure of this proof of Lemma 7.4.3 is outlined in Fig. 7.4.

For all tasks t and states σ such that $t, \sigma \approx^* \overline{\tilde{v}, \tilde{I}, \Phi}$, we have by definition of simulation ($\approx^*$) that we know that for each tuple $(\tilde{v}, \tilde{I}, \Phi)$, the following sequence of symbolic drive steps has occurred.

$$\begin{aligned} t, \sigma &\approx \tilde{t}_1, \tilde{\sigma}_1, \tilde{i}_1, \phi_1 \\ &\quad \tilde{t}_1, \tilde{\sigma}_1 \approx \quad \quad \quad \tilde{t}_2, \tilde{\sigma}_2, \tilde{i}_2, \phi_2 \\ &\quad \quad \quad \quad \quad \quad \tilde{t}_2, \tilde{\sigma}_2 \approx \dots \\ &\quad \quad \quad \quad \quad \quad \dots \quad \quad \quad \approx \tilde{t}_n, \tilde{\sigma}_n, \tilde{i}_n, \phi_n \end{aligned}$$

with $\mathcal{V}(\tilde{t}_n, \tilde{\sigma}_n) = \tilde{v}$ and $\mathcal{S}(\phi_1 \wedge \dots \wedge \phi_n)$.

We need to show that there exists an I such that $t, \sigma \xRightarrow{I}^* v$, which is defined similarly as follows.

$$t, \sigma \xRightarrow{i_1} t_1, \sigma_1 \xRightarrow{i_2} t_2, \sigma_2 \xRightarrow{i_3} \dots \xRightarrow{i_n} t_n, \sigma_n \text{ with } \mathcal{V}(t_n, \sigma_n).$$

By Lemma 7.4.6, we know that there exists an i_1 such that $t, \sigma \xRightarrow{i_1} t_1, \sigma_1$, since $t, \sigma \sqsubseteq_{\mathcal{O}} t, \sigma, \text{True}$. This also gives us that $\tilde{i}_1 \sim i_1$, and $t_1, \sigma_1 \sqsubseteq_{[s_1 \mapsto c_1]} \tilde{t}_1, \tilde{\sigma}_1, \phi_1$ with $\text{SymOf}(\tilde{i}_1) = s_1$ and $\text{ValOf}(i_1) = c_1$.

By repeatedly applying Lemma 7.4.6, until we arrive at $\tilde{t}_n, \tilde{\sigma}_n$, we can show that there indeed exists an input list I such that $t, \sigma \xRightarrow{I}^* v$ with $[s_1 \mapsto c_1, \dots, s_n \mapsto c_n] \tilde{v} = v$ and $[s_1 \mapsto c_1, \dots, s_n \mapsto c_n] \Phi$, namely $I = [i_1, \dots, i_n]$. $\square$

E.2.2 Soundness of interaction

Proof: The proof of Lemma 7.4.6 only consists of one case, since the symbolic interacting semantics consists of only one rule, SI-HANDLE. Given that $t, \sigma \sqsubseteq_M \tilde{t}, \tilde{\sigma}, \Phi$ and $\tilde{t}, \tilde{\sigma} \approx \overline{\tilde{t}', \tilde{\sigma}', \tilde{i}, \phi_1}$, Lemma E.2.1 gives us that for each tuple $(\tilde{t}', \tilde{\sigma}', \tilde{i}, \phi_1)$ there exists an input i such that $\tilde{i} \sim i$, $t, \sigma \xrightarrow{i} t', \sigma'$ and $t', \sigma' \sqsubseteq_{[s \mapsto c]M} \tilde{t}', \tilde{\sigma}', \Phi \wedge \phi_1$ with $s = \text{SymOf}(\tilde{i})$ and $c = \text{ValOf}(i)$.

Then, by Lemma E.2.2, given that $\tilde{t}', \tilde{\sigma}' \ll \overline{\tilde{t}'', \tilde{\sigma}'', \phi_2}$, we obtain that for each tuple $(\tilde{t}'', \tilde{\sigma}'', \phi_2)$, we have that $\mathcal{S}(\Phi \wedge \phi_1 \wedge \phi_2)$ implies that $t', \sigma' \Downarrow t'', \sigma''$ with $t'', \sigma'' \sqsubseteq_{[s \mapsto c]M} \tilde{t}'', \tilde{\sigma}'', \Phi \wedge \phi_1 \wedge \phi_2$.

The above, together with the I-HANDLE gives us that there exists an input i such that $t, \sigma \xrightarrow{i} t'', \sigma''$. $\square$

E.2.3 Soundness of handle

Lemma E.2.1 (Soundness of handle)

For all concrete tasks t , concrete states σ , symbolic tasks $\tilde{t}$, symbolic states $\tilde{\sigma}$ path conditions Φ and mappings M , we have that $t, \sigma \sqsubseteq_M \tilde{t}, \tilde{\sigma}, \Phi$ implies that for all symbolic inputs $\tilde{i}$ such that $\tilde{t}, \tilde{\sigma} \rightsquigarrow \tilde{t}', \tilde{\sigma}', \tilde{i}, \phi$ and for all tuples $(\tilde{t}', \tilde{\sigma}', \tilde{i}, \phi)$, $\mathcal{S}(\Phi \wedge \phi)$ implies that there exists an input i such that $\tilde{i} \sim i$, $t, \sigma \xrightarrow{i} t', \sigma'$ and $t', \sigma' \sqsubseteq_{[s \mapsto c]M} \tilde{t}', \tilde{\sigma}', \Phi \wedge \phi$ where $\text{SymOf}(\tilde{i}) = s$ and $\text{ValOf}(i) = c$.

Proof: We prove Lemma E.2.1 by induction over the derivation $\tilde{t}, \tilde{\sigma} \rightsquigarrow \tilde{t}', \tilde{\sigma}', \tilde{i}, \phi$.

Case SH-FILL

We have $t, \sigma \sqsubseteq_M \boxtimes \beta, \tilde{\sigma}, \Phi$, since we know from SH-FILL that $\tilde{t} = \boxtimes \beta$. From the consistency relation, we know that $t = M(\boxtimes \beta)$. Since $\boxtimes \beta$ does not contain any symbols, we know that t must be $\boxtimes \beta$ too. There exists only one symbolic execution, namely $\boxtimes \beta, \tilde{\sigma} \rightsquigarrow \square s, \tilde{\sigma}, s, \text{True}$ with $s : \beta$. We need to show that there exists an i such that $s \sim i$ and $\square v, \sigma \xrightarrow{i} t', \sigma'$, by H-FILL.

Any concrete value c of type β will do. Now we have to show that we end up with $\square c, \sigma \sqsubseteq_{[s \mapsto c]M} \square s, \tilde{\sigma}, \Phi \wedge \text{True}$, which holds trivially.

Case SH-CHANGE

Since we have $t, \sigma \sqsubseteq_M \square \tilde{v}, \tilde{\sigma}, \Phi$ and $\square \tilde{v}, \tilde{\sigma} \rightsquigarrow \square s, \tilde{\sigma}, s, \text{True}$ with $\tilde{v}, s : \beta$, we know that either $\tilde{v}$ is a concrete value, or M contains a mapping such that $M\tilde{v}$ becomes a concrete value c . We know therefore that t must be $\square c$ with $c : \beta$.

We need to show that there exists an i such that $s \sim i$ and $\square c, \sigma \xrightarrow{i} t', \sigma'$ by H-CHANGE.

Any concrete value c' of the same type as c will do. Now we have to show that we end up with $\square c', \sigma \sqsubseteq_{[s \mapsto c']M} \square s, \tilde{\sigma}, \Phi \wedge \text{True}$, which holds trivially.

Case SH-UPDATE

Since we have $t, \sigma \sqsubseteq_M \blacksquare l, \tilde{\sigma}, \Phi$ and $\blacksquare l, \tilde{\sigma} \rightsquigarrow \blacksquare l, \tilde{\sigma}[l \mapsto s], s, \text{True}$ with $\tilde{\sigma}(l), s : \beta$, we know that t must be $\blacksquare l$ too, $\tilde{t}$ contains no symbols. We need to show that there exists an i such that $s \sim i$ and $\blacksquare l, \sigma \xrightarrow{i} t', \sigma'$ by H-UPDATE.

Any concrete value c of the same type as l will do. Now we have to show that we end up with $\blacksquare l, \sigma[l \mapsto c] \sqsubseteq_{[s \mapsto c]M} \blacksquare l, \tilde{\sigma}[l \mapsto s], \Phi \wedge \text{True}$, which holds trivially.

Case SH-NEXT

Since we have $t, \sigma \sqsubseteq_M \tilde{t}_1 \triangleright \tilde{e}_2, \tilde{\sigma}, \Phi$ and $\tilde{t}_1 \triangleright \tilde{e}_2, \tilde{\sigma} \rightsquigarrow \overline{\tilde{t}'_1 \triangleright \tilde{e}_2, \tilde{\sigma}_1, \tilde{i}, \phi_1} \cup \overline{\tilde{t}_2, \tilde{\sigma}_2, C, \phi_2}$, we know that $M\tilde{t}_1 \triangleright \tilde{e}_2 = t$, which comes down to $t_1 \triangleright e_2$ for some concrete t_1 and e_2 .

In this case, we have two sets of symbolic executions.

For all tuples $(\tilde{t}'_1 \triangleright \tilde{e}_2, \tilde{\sigma}'_1, \tilde{i}, \phi_1)$, we know by application of the induction hypothesis that there exists an i such that $\tilde{i} \sim i$, $t_1, \sigma \xrightarrow{i} t'_1, \sigma'$ and $t'_1, \sigma' \sqsubseteq_{[s \mapsto c]M} \tilde{t}'_1, \tilde{\sigma}', \Phi \wedge \phi_1$ where $\text{ValOf}(i) = c$ and $\text{ValOf}(\tilde{i}) = s$. Therefore we also have $t'_1 \triangleright e_2, \sigma'_1 \sqsubseteq_{[s \mapsto c]M} \tilde{t}'_1 \triangleright \tilde{e}_2, \tilde{\sigma}'_1, \Phi \wedge \phi_1$.

For all tuples $(\tilde{t}_2, \tilde{\sigma}'_2, C, \phi_2)$, we first have by Lemma E.2.5 that $v_1, \sigma \sqsubseteq_M \tilde{v}_1, \tilde{\sigma}, \Phi$. Now, before we can apply Lemma E.2.2, we need to establish that $e_2 v_1, \sigma \sqsubseteq_M \tilde{e}_2 \tilde{v}_1, \tilde{\sigma}, \Phi$ holds. This means that we have to show that $M\tilde{e}_2 \tilde{v}_1 = e_2 v_1$. Since application of the mapping is distributive, it suffices to show that $M\tilde{v}_1 = v_1$, which is given, and $M\tilde{e}_2 = e_2$, which follows from the premise as well.

At this point, by application of Lemma E.2.2, we obtain that $e_2 v_1, \sigma \Downarrow t_2, \sigma'_2$ and $t_2, \sigma'_2 \sqsubseteq_M \tilde{t}_2, \tilde{\sigma}'_2, \Phi \wedge \phi_2$. This, together with the H-NEXT rule leads us to conclude that this case holds as well.

Case SH-PASSNEXT, SH-PASSNEXTFAIL

Since we have $t, \sigma \sqsubseteq_M \tilde{t}_1 \triangleright \tilde{e}_2, \tilde{\sigma}, \Phi$ and $\tilde{t}_1 \triangleright \tilde{e}_2, \tilde{\sigma} \rightsquigarrow \overline{\tilde{t}'_1 \triangleright \tilde{e}_2, \tilde{\sigma}', \tilde{i}, \phi}$, we know that $M\tilde{t}_1 \triangleright \tilde{e}_2 = t$, which comes down to $t_1 \triangleright e_2$ for some concrete t_1 and e_2 .

For all tuples $(\tilde{t}'_1 \triangleright \tilde{e}_2, \tilde{\sigma}'_1, \tilde{i}, \phi_1)$, we know by application of the induction hypothesis that there exists an i such that $\tilde{i} \sim i$, $t_1, \sigma \xrightarrow{i} t'_1, \sigma'$ and $t'_1, \sigma' \sqsubseteq_{[s \mapsto c]M} \tilde{t}'_1, \tilde{\sigma}', \Phi \wedge \phi_1$ where $\text{ValOf}(i) = c$ and $\text{ValOf}(\tilde{i}) = s$. Therefore we also have $t'_1 \triangleright e_2, \sigma'_1 \sqsubseteq_{[s \mapsto c]M} \tilde{t}'_1 \triangleright \tilde{e}_2, \tilde{\sigma}'_1, \Phi \wedge \phi_1$.

Case SH-PASSTHEN

Since we have $t, \sigma \sqsubseteq_M \tilde{t}_1 \blacktriangleright \tilde{e}_2, \tilde{\sigma}, \Phi$ and $\tilde{t}_1 \blacktriangleright \tilde{e}_2, \tilde{\sigma} \rightsquigarrow \overline{\tilde{t}'_1 \blacktriangleright \tilde{e}_2, \tilde{\sigma}', \tilde{i}, \phi}$, we know that $M\tilde{t}_1 \blacktriangleright \tilde{e}_2 = t$, which comes down to $t_1 \blacktriangleright e_2$ for some concrete t_1 and e_2 .

For all tuples $(\tilde{t}'_1 \blacktriangleright \tilde{e}_2, \tilde{\sigma}', \tilde{i}, \phi)$, we know by application of the induction hypothesis that there exists an i such that $\tilde{i} \sim i$, $t_1, \sigma \xrightarrow{i} t'_1, \sigma'$ and $t'_1, \sigma' \sqsubseteq_{[s \mapsto c]M} \tilde{t}'_1, \tilde{\sigma}', \Phi \wedge \phi_1$ where $\text{ValOf}(i) = c$ and $\text{SymOf}(\tilde{i}) = s$. Therefore we also have $t'_1 \blacktriangleright e_2, \sigma'_1 \sqsubseteq_{[s \mapsto c]M} \tilde{t}'_1 \blacktriangleright \tilde{e}_2, \tilde{\sigma}'_1, \Phi \wedge \phi_1$.

Case SH-PICK

In this case, we have two sets of symbolic executions.

For all tuples $(\tilde{t}_1, \tilde{\sigma}_1, L, \phi_1)$, we obtain from Lemma E.2.2 that $e_1, \sigma \Downarrow t_1, \sigma_1$ with $t_1, \sigma_1 \Leftarrow_M \tilde{t}_1, \tilde{\sigma}_1, \Phi \wedge \phi_1$.

For all tuples $(\tilde{t}_2, \tilde{\sigma}_2, R, \phi_2)$, we obtain from Lemma E.2.2 that $e_2, \sigma \Downarrow t_2, \sigma_2$ with $t_2, \sigma_2 \Leftarrow_M \tilde{t}_2, \tilde{\sigma}_2, \Phi \wedge \phi_2$.

Case SH-PICKLEFT

For all tuples $(\tilde{t}_1, \tilde{\sigma}_1, L, \phi_1)$, we obtain from Lemma E.2.2 that $e_1, \sigma \Downarrow t_1, \sigma_1$ with $t_1, \sigma_1 \Leftarrow_M \tilde{t}_1, \tilde{\sigma}_1, \Phi \wedge \phi_1$. This, together with the H-PICKLEFT rule leads us to conclude that this case holds as well.

Case SH-PICKRIGHT

For all tuples $(\tilde{t}_2, \tilde{\sigma}_2, R, \phi_2)$, we obtain from Lemma E.2.2 that $e_2, \sigma \Downarrow t_2, \sigma_2$ with $t_2, \sigma_2 \Leftarrow_M \tilde{t}_2, \tilde{\sigma}_2, \Phi \wedge \phi_2$. This, together with the H-PICKRIGHT rule leads us to conclude that this case holds as well.

Case SH-AND

We have that $t, \sigma \Leftarrow_M \tilde{t}_1 \bowtie \tilde{t}_2, \tilde{\sigma}, \Phi$ and $\tilde{t}_1 \bowtie \tilde{t}_2, \tilde{\sigma} \rightsquigarrow \overline{\tilde{t}'_1 \bowtie \tilde{t}_2, \tilde{\sigma}'_1, F \tilde{t}_1, \phi_1} \cup \tilde{t}_1 \bowtie \tilde{t}'_2, \tilde{\sigma}'_2, S \tilde{t}_2, \phi_2$.

In this case, we have two sets of symbolic executions.

For all tuples $(\tilde{t}'_1 \bowtie \tilde{t}_2, \tilde{\sigma}'_1, F \tilde{t}_1, \phi_1)$, we know by application of the induction hypothesis that there exists an i such that $\tilde{t}_1 \sim i, t_1, \sigma \xrightarrow{i} t'_1, \sigma'_1$ and $t'_1, \sigma'_1 \Leftarrow_{[s \mapsto c]M} \tilde{t}'_1, \tilde{\sigma}'_1, \Phi \wedge \phi_1$. Then by H-FIRSTAND, we know that also $t_1 \bowtie t_2, \sigma \xrightarrow{Fi} t'_1 \bowtie t_2, \sigma'_1$. It follows trivially that $t'_1 \bowtie t_2, \sigma'_1 \Leftarrow_{[s \mapsto c]M} \tilde{t}'_1 \bowtie \tilde{t}_2, \tilde{\sigma}'_1, \Phi \wedge \phi_1$.

For all tuples $(\tilde{t}_1 \bowtie \tilde{t}'_2, \tilde{\sigma}'_2, S \tilde{t}_2, \phi_2)$, we know by application of the induction hypothesis that there exists an i such that $\tilde{t}_2 \sim i, t_2, \sigma \xrightarrow{i} t'_2, \sigma'_2$ and $t'_2, \sigma'_2 \Leftarrow_{[s \mapsto c]M} \tilde{t}'_2, \tilde{\sigma}'_2, \Phi \wedge \phi_2$. Then by H-SECONDAND, we know that also $t_1 \bowtie t_2, \sigma \xrightarrow{Si} t_1 \bowtie t'_2, \sigma'_2$. It follows trivially that $t_1 \bowtie t'_2, \sigma'_2 \Leftarrow_{[s \mapsto c]M} \tilde{t}_1 \bowtie \tilde{t}'_2, \tilde{\sigma}'_2, \Phi \wedge \phi_2$.

Case SH-OR

This case is proven in the same way as the case for the SH-AND rule. $\square$

E.2.4 Soundness of normalise

Lemma E.2.2 (Soundness of normalisation)

For all concrete expressions e , concrete states σ , symbolic expressions $\tilde{e}$, symbolic states $\tilde{\sigma}$ path conditions Φ and mappings M , we have that $e, \sigma \Leftarrow_M \tilde{e}, \tilde{\sigma}, \Phi$ implies that if $\tilde{e}, \tilde{\sigma} \Downarrow \tilde{t}, \tilde{\sigma}', \phi$, then for all tuples $(\tilde{t}, \tilde{\sigma}', \phi)$ it holds that $\mathcal{S}(\Phi \wedge \phi)$ implies that $e, \sigma \Downarrow t, \sigma'$ with $t, \sigma' \Leftarrow_M \tilde{t}, \tilde{\sigma}', \Phi \wedge \phi$.

Proof: We prove Lemma E.2.2 by induction over the derivation $\tilde{e}, \tilde{\sigma} \Downarrow \tilde{t}, \tilde{\sigma}', \phi$.

From the premise, we can assume that $e, \sigma \sqsubseteq_M \tilde{e}, \tilde{\sigma}, \Phi$. Now, given that $\tilde{e}, \tilde{\sigma} \Downarrow \tilde{t}, \tilde{\sigma}', \phi$, we need to demonstrate that for all tuples $(\tilde{t}, \tilde{\sigma}', \phi)$, $\mathcal{S}(\Phi \wedge \phi)$ implies that $e, \sigma \Downarrow t, \sigma'$ with $t, \sigma' \sqsubseteq_M \tilde{t}, \tilde{\sigma}', \Phi \wedge \phi$.

The base case is when the SN-DONE rule applies.

In this case, we obtain from Lemma E.2.4 that $e, \sigma \Downarrow t, \sigma'$ with $t, \sigma' \sqsubseteq_M \tilde{t}, \tilde{\sigma}', \Phi \wedge \phi_1$. In order to show that the N-DONE rule applies, we need to additionally show that $t, \sigma' \mapsto t', \sigma''$. This is directly obtained from Lemma E.2.3.

The only induction step is when SN-REPEAT applies.

In this case, we obtain from Lemma E.2.4 that $e, \sigma \Downarrow t, \sigma'$ with $t, \sigma' \sqsubseteq_M \tilde{t}, \tilde{\sigma}', \Phi \wedge \phi_1$, which is exactly what we need to show. Furthermore, by Lemma E.2.3 we obtain that $t, \sigma' \mapsto t', \sigma''$ with $t', \sigma'' \sqsubseteq_M \tilde{t}', \tilde{\sigma}'', \Phi \wedge \phi_1 \wedge \phi_2$. Then finally, by application of the induction hypothesis, we obtain what we need to prove: $t', \sigma'' \Downarrow t'', \sigma'''$ with $t'', \sigma''' \sqsubseteq_M \tilde{t}'', \tilde{\sigma}''', \Phi \wedge \phi_1 \wedge \phi_2 \wedge \phi_3$. $\square$

E.2.5 Soundness of stride

Lemma E.2.3 (Soundness of stride)

For all concrete tasks t , concrete states σ , symbolic tasks $\tilde{t}$, symbolic states $\tilde{\sigma}$, path conditions Φ and mappings M , we have that $t, \sigma \sqsubseteq_M \tilde{t}, \tilde{\sigma}, \Phi$ implies that if $\tilde{t}, \tilde{\sigma} \rightsquigarrow \tilde{t}', \tilde{\sigma}', \phi$, then for all tuples $(\tilde{t}', \tilde{\sigma}', \phi)$ it holds that $\mathcal{S}(\Phi \wedge \phi)$ implies that $t, \sigma \mapsto t', \sigma'$ with $t', \sigma' \sqsubseteq_M \tilde{t}', \tilde{\sigma}', \Phi \wedge \phi$.

Proof: Provided that $t, \sigma \sqsubseteq_M \tilde{t}, \tilde{\sigma}, \Phi$ and $\tilde{t}, \tilde{\sigma} \rightsquigarrow \tilde{t}', \tilde{\sigma}', \phi$, we want to show that for all tuples $(\tilde{t}', \tilde{\sigma}', \phi)$, we have $\mathcal{S}(\Phi \wedge \phi)$ implies that $t, \sigma \mapsto t', \sigma'$. We prove Lemma E.2.3 by induction over the derivation $\tilde{t}, \tilde{\sigma} \rightsquigarrow \tilde{t}', \tilde{\sigma}', \phi$.

Case SS-EDIT

Given that $t, \sigma \sqsubseteq_M \Box \tilde{v}, \tilde{\sigma}, \Phi$ and $\Box \tilde{v}, \tilde{\sigma} \rightsquigarrow \Box \tilde{v}, \tilde{\sigma}, \text{True}$, we know that $t = \Box M\tilde{v}$, and we have $\Box M\tilde{v}, \sigma \mapsto \Box M\tilde{v}, \sigma$ by S-EDIT and $\Box M\tilde{v}, \sigma \sqsubseteq_M \Box \tilde{v}, \tilde{\sigma}, \Phi$, since none of the tasks and states were altered.

Case SS-FILL, SS-UPDATE, SS-FAIL, SS-XOR

These cases are proven in the same way as the case for SS-EDIT.

Case SS-THENSTAY

Provided that $t, \sigma \sqsubseteq_M \tilde{t}_1 \blacktriangleright \tilde{e}_2, \tilde{\sigma}, \Phi$ and $\tilde{t}_1 \blacktriangleright \tilde{e}_2, \tilde{\sigma} \rightsquigarrow \tilde{t}'_1 \blacktriangleright \tilde{e}_2, \tilde{\sigma}', \phi$, we obtain from the induction hypothesis that $t_1, \sigma \mapsto t'_1, \sigma'$ and $t'_1, \sigma' \sqsubseteq_M$

$\tilde{t}'_1, \tilde{\sigma}', \Phi$. From this, we can directly conclude that $t_1 \blacktriangleright e_2, \sigma \mapsto t'_1 \blacktriangleright e_2, \sigma'$ and $t'_1 \blacktriangleright e_2, \sigma' \sqsubseteq_M \tilde{t}'_1 \blacktriangleright \tilde{e}_2, \tilde{\sigma}', \Phi$.

Case SS-THENFAIL

Provided that $t, \sigma \sqsubseteq_M \tilde{t}_1 \blacktriangleright \tilde{e}_2, \tilde{\sigma}, \Phi$ and $\tilde{t}_1 \blacktriangleright \tilde{e}_2, \tilde{\sigma} \rightsquigarrow \tilde{t}'_1 \blacktriangleright \tilde{e}_2, \tilde{\sigma}', \phi$, we obtain from the induction hypothesis that $t_1, \sigma \mapsto t'_1, \sigma'$ and $t'_1, \sigma' \sqsubseteq_M \tilde{t}'_1, \tilde{\sigma}', \Phi$. From this, we can directly conclude that $t_1 \blacktriangleright e_2, \sigma \mapsto t'_1 \blacktriangleright e_2, \sigma'$ and $t'_1 \blacktriangleright e_2, \sigma' \sqsubseteq_M \tilde{t}'_1 \blacktriangleright \tilde{e}_2, \tilde{\sigma}', \Phi$.

Case SS-THENCONT

Provided that $t, \sigma \sqsubseteq_M \tilde{t}_1 \blacktriangleright \tilde{e}_2, \tilde{\sigma}, \Phi$ and $\tilde{t}_1 \blacktriangleright \tilde{e}_2, \tilde{\sigma} \rightsquigarrow \tilde{t}_2, \tilde{\sigma}', \phi_1 \wedge \phi_2$ with $\tilde{t}_1, \tilde{\sigma} \rightsquigarrow \tilde{t}'_1, \tilde{\sigma}', \phi_1$ and $\mathcal{V}(\tilde{t}'_1, \tilde{\sigma}') = \tilde{v}_1$, we obtain from the induction hypothesis that $t_1, \sigma \mapsto t'_1, \sigma'$ and $t'_1, \sigma' \sqsubseteq_M \tilde{t}'_1, \tilde{\sigma}', \Phi$. From Lemma E.2.5 we know that the value function preserves the consistency relation, so we have that $v_1, \sigma' \sqsubseteq_M \tilde{v}_1, \tilde{\sigma}', \Phi$ with $\mathcal{V}(t'_1, \sigma') = v_1$.

At this point, we have $e_2 v_1, \sigma' \sqsubseteq_M \tilde{e}_2 \tilde{v}_1, \tilde{\sigma}', \Phi \wedge \phi_1$ and $\tilde{e}_2 \tilde{v}_1, \tilde{\sigma}' \Downarrow \tilde{t}_2, \tilde{\sigma}'', \phi_2$. This allows us to apply Lemma E.2.4 to obtain $e_2 v_1, \sigma' \Downarrow t_2, \sigma''$ and $t_2, \sigma'' \sqsubseteq_M \tilde{t}_2, \tilde{\sigma}'', \Phi \wedge \phi_1 \wedge \phi_2$.

From this, we can directly conclude that $t_1 \blacktriangleright e_2, \sigma \mapsto t_2, \sigma''$ and $t_2, \sigma'' \sqsubseteq_M \tilde{t}_2, \tilde{\sigma}'', \Phi \wedge \phi_1 \wedge \phi_2$.

Case SS-ORLEFT

Provided that $t, \sigma \sqsubseteq_M \tilde{t}_1 \blacklozenge \tilde{t}_2, \tilde{\sigma}, \Phi$ and $\tilde{t}_1 \blacklozenge \tilde{t}_2, \tilde{\sigma} \rightsquigarrow \tilde{t}'_1, \tilde{\sigma}', \phi$, we obtain from the induction hypothesis that $t_1, \sigma \mapsto t'_1, \sigma'$ and $t'_1, \sigma' \sqsubseteq_M \tilde{t}'_1, \tilde{\sigma}', \Phi \wedge \phi$. From this, we can directly conclude that $t_1 \blacklozenge t_2, \sigma \mapsto t'_1, \sigma'$.

Case SS-ORRIGHT

Provided that $t, \sigma \sqsubseteq_M \tilde{t}_1 \blacklozenge \tilde{t}_2, \tilde{\sigma}, \Phi$ and $\tilde{t}_1 \blacklozenge \tilde{t}_2, \tilde{\sigma} \rightsquigarrow \tilde{t}'_2, \tilde{\sigma}'', \phi_1 \wedge \phi_2$, we obtain from the induction hypothesis that $t_1, \sigma \mapsto t'_1, \sigma'$ and $t'_1, \sigma' \sqsubseteq_M \tilde{t}'_1, \tilde{\sigma}', \Phi \wedge \phi_1$. Then by a second application of the induction hypothesis, we obtain that $t_2, \sigma' \mapsto t'_2, \sigma''$ and $t'_2, \sigma'' \sqsubseteq_M \tilde{t}'_2, \tilde{\sigma}'', \Phi \wedge \phi_1 \wedge \phi_2$. This leads us to conclude $t_1 \blacklozenge t_2, \sigma \mapsto t'_1 \blacklozenge t'_2, \sigma''$.

Case SS-ORNONE

Provided that $t, \sigma \sqsubseteq_M \tilde{t}_1 \blacklozenge \tilde{t}_2, \tilde{\sigma}, \Phi$ and $\tilde{t}_1 \blacklozenge \tilde{t}_2, \tilde{\sigma} \rightsquigarrow \tilde{t}'_2, \tilde{\sigma}'', \phi_1 \wedge \phi_2$, we obtain from the induction hypothesis that $t_1, \sigma \mapsto t'_1, \sigma'$ and $t'_1, \sigma' \sqsubseteq_M \tilde{t}'_1, \tilde{\sigma}', \Phi \wedge \phi_1$. Then by a second application of the induction hypothesis, we obtain that $t_2, \sigma' \mapsto t'_2, \sigma''$ and $t'_2, \sigma'' \sqsubseteq_M \tilde{t}'_2, \tilde{\sigma}'', \Phi \wedge \phi_1 \wedge \phi_2$. This leads us to conclude $t_1 \blacklozenge t_2, \sigma \mapsto t'_1 \blacklozenge t'_2, \sigma''$ and $t'_1 \blacklozenge t'_2, \sigma'' \sqsubseteq_M \tilde{t}'_1 \blacklozenge \tilde{t}'_2, \tilde{\sigma}'', \Phi \wedge \phi_1 \wedge \phi_2$.

Case SS-NEXT

Provided that $t, \sigma \sqsubseteq_M \tilde{t}_1 \triangleright \tilde{e}_2, \tilde{\sigma}, \Phi$ and $\tilde{t}_1 \triangleright \tilde{e}_2, \tilde{\sigma} \rightsquigarrow \tilde{t}'_1 \blacktriangleright \tilde{e}_2, \tilde{\sigma}', \phi$, we obtain from the induction hypothesis that $t_1, \sigma \mapsto t'_1, \sigma'$ and $t'_1, \sigma' \sqsubseteq_M \tilde{t}'_1, \tilde{\sigma}', \Phi \wedge \phi$. From this, we can directly conclude that $t_1 \triangleright e_2, \sigma \mapsto t'_1 \triangleright e_2, \sigma'$ and $t'_1 \triangleright e_2, \sigma' \sqsubseteq_M \tilde{t}'_1 \triangleright \tilde{e}_2, \tilde{\sigma}', \Phi \wedge \phi$.

Case SS-AND

Provided that $t, \sigma \sqsubseteq_M \tilde{t}_1 \bowtie \tilde{t}_2, \tilde{\sigma}, \Phi$ and $\tilde{t}_1 \bowtie \tilde{t}_2, \tilde{\sigma} \rightsquigarrow \tilde{t}'_1 \bowtie \tilde{t}'_2, \tilde{\sigma}'', \phi_1 \wedge \phi_2$, we obtain from the induction hypothesis that $t_1, \sigma \mapsto t'_1, \sigma'$ and $t'_1, \sigma' \sqsubseteq_M \tilde{t}'_1, \tilde{\sigma}', \Phi \wedge \phi_1$. Then by a second application of the induction hypothesis, we obtain that $t_2, \sigma' \mapsto t'_2, \sigma''$ and $t'_2, \sigma'' \sqsubseteq_M \tilde{t}'_2, \tilde{\sigma}'', \Phi \wedge \phi_1 \wedge \phi_2$. This leads us to conclude $t_1 \bowtie t_2, \sigma \mapsto t'_1 \bowtie t'_2, \sigma''$ and $t'_1 \bowtie t'_2, \sigma'' \sqsubseteq_M \tilde{t}'_1 \bowtie \tilde{t}'_2, \tilde{\sigma}'', \Phi \wedge \phi_1 \wedge \phi_2$.

□

E.2.6 Soundness of evaluate**Lemma E.2.4** (Soundness of evaluate)

For all concrete expressions e , concrete states σ , symbolic expressions $\tilde{e}$, symbolic states $\tilde{\sigma}$, path conditions Φ and mappings M , we have that $e, \sigma \sqsubseteq_M \tilde{e}, \tilde{\sigma}, \Phi$ implies that if $\tilde{e}, \tilde{\sigma} \Downarrow \tilde{v}, \tilde{\sigma}', \phi$, then for all tuples $(\tilde{v}, \tilde{\sigma}', \phi)$ it holds that $S(\Phi \wedge \phi)$ implies that $e, \sigma \Downarrow v, \sigma'$ with $v, \sigma' \sqsubseteq_M \tilde{v}, \tilde{\sigma}', \Phi \wedge \phi$.

Proof: We prove Lemma E.2.4 by induction over the derivation $\tilde{e}, \tilde{\sigma} \Downarrow \tilde{v}, \tilde{\sigma}', \phi$.

Case SE-VALUE

We assume $e, \sigma \sqsubseteq_M \tilde{v}, \tilde{\sigma}, \Phi$ and $\tilde{v}, \tilde{\sigma} \Downarrow \tilde{v}, \tilde{\sigma}, \text{True}$. By E-VALUE we have $v, \sigma \Downarrow v, \sigma$, so this case holds trivially.

Case SE-PAIR

Provided that $e, \sigma \sqsubseteq_M \langle \tilde{e}_1, \tilde{e}_2 \rangle, \tilde{\sigma}, \Phi$ and $\langle \tilde{e}_1, \tilde{e}_2 \rangle, \tilde{\sigma} \Downarrow \langle \tilde{v}_1, \tilde{v}_2 \rangle, \tilde{\sigma}'', \phi_1 \wedge \phi_2$, we know that $e = \langle e_1, e_2 \rangle$. We obtain from the induction hypothesis that $e_1, \sigma \Downarrow v_1, \sigma'$ with $v_1, \sigma' \sqsubseteq_M \tilde{v}_1, \tilde{\sigma}', \Phi \wedge \phi_1$. Then by a second application of the induction hypothesis, we obtain that $e_2, \sigma' \Downarrow v_2, \sigma''$ with $v_2, \sigma'' \sqsubseteq_M \tilde{v}_2, \tilde{\sigma}'', \Phi \wedge \phi_1 \wedge \phi_2$. From this, we can conclude that $\langle e_1, e_2 \rangle, \sigma \Downarrow \langle v_1, v_2 \rangle, \sigma''$ with $\langle v_1, v_2 \rangle, \sigma'' \sqsubseteq_M \langle \tilde{v}_1, \tilde{v}_2 \rangle, \tilde{\sigma}'', \Phi \wedge \phi_1 \wedge \phi_2$.

Case SE-FIRST

Provided that $e, \sigma \sqsubseteq_M \text{fst } \tilde{e}, \tilde{\sigma}, \Phi$ and $\text{fst } \tilde{e}, \tilde{\sigma} \Downarrow \tilde{v}_1, \tilde{\sigma}'', \phi$, we know that $e = \text{fst } e$. We obtain from the induction hypothesis that $e, \sigma \Downarrow \langle v_1, v_2 \rangle, \sigma'$

with $\langle v_1, v_2 \rangle, \sigma' \sqsubseteq_M \langle \tilde{v}_1, \tilde{v}_2 \rangle, \tilde{\sigma}', \Phi \wedge \phi$. From this, we can conclude that $\text{fst } e, \sigma \downarrow v_1, \sigma'$ and $v_1, \sigma' \sqsubseteq_M \tilde{v}_1, \tilde{\sigma}', \Phi \wedge \phi$.

Case SE-SECOND

Provided that $e, \sigma \sqsubseteq_M \text{snd } \tilde{e}, \tilde{\sigma}, \Phi$ and $\text{snd } \tilde{e}, \tilde{\sigma} \downarrow \tilde{v}_2, \tilde{\sigma}'', \phi$, we know that $e = \text{snd } e$. We obtain from application of the induction hypothesis that $e, \sigma \downarrow \langle v_1, v_2 \rangle, \sigma'$ with $\langle v_1, v_2 \rangle, \sigma' \sqsubseteq_M \langle \tilde{v}_1, \tilde{v}_2 \rangle, \tilde{\sigma}', \Phi \wedge \phi$. From this, we can conclude that $\text{snd } e, \sigma \downarrow v_2, \sigma'$ and $v_2, \sigma' \sqsubseteq_M \tilde{v}_2, \tilde{\sigma}', \Phi \wedge \phi$.

Case SE-CONS

Provided that $e, \sigma \sqsubseteq_M \tilde{e}_1 :: \tilde{e}_2, \tilde{\sigma}, \Phi$ and $\tilde{e}_1 :: \tilde{e}_2, \tilde{\sigma} \downarrow \tilde{v}_1 :: \tilde{v}_2, \tilde{\sigma}'', \phi_1 \wedge \phi_2$, we know that $e = e_1 :: e_2$. We obtain from the induction hypothesis that $e_1, \sigma \downarrow v_1, \sigma'$ with $v_1, \sigma' \sqsubseteq_M \tilde{v}_1, \tilde{\sigma}', \Phi \wedge \phi_1$. Then by a second application of the induction hypothesis, we obtain that $e_2, \sigma' \downarrow v_2, \sigma''$ with $v_2, \sigma'' \sqsubseteq_M \tilde{v}_2, \tilde{\sigma}'', \Phi \wedge \phi_1 \wedge \phi_2$. From this, we can conclude that $e_1 :: e_2, \sigma \downarrow v_1 :: v_2, \sigma''$ with $v_1 :: v_2, \sigma'' \sqsubseteq_M \tilde{v}_1 :: \tilde{v}_2, \tilde{\sigma}'', \Phi \wedge \phi_1 \wedge \phi_2$.

Case SE-HEAD

Provided that $e, \sigma \sqsubseteq_M \text{head } \tilde{e}, \tilde{\sigma}, \Phi$ and $\text{head } \tilde{e}, \tilde{\sigma} \downarrow \tilde{v}_1, \tilde{\sigma}', \phi$, we know that $e = \text{head } e$. We obtain from the induction hypothesis that $e, \sigma \downarrow v_1 :: v_2, \sigma'$ with $v_1 :: v_2, \sigma' \sqsubseteq_M \tilde{v}_1 :: \tilde{v}_2, \tilde{\sigma}', \Phi \wedge \phi$. From this, we can conclude that $\text{head } e, \sigma \downarrow v_1, \sigma'$.

Case SE-TAIL

Provided that $e, \sigma \sqsubseteq_M \text{tail } \tilde{e}, \tilde{\sigma}, \Phi$ and $\text{tail } \tilde{e}, \tilde{\sigma} \downarrow \tilde{v}_2, \tilde{\sigma}', \phi$, we know that $e = \text{tail } e$. We obtain from the induction hypothesis that $e, \sigma \downarrow v_1 :: v_2, \sigma'$ with $v_1 :: v_2, \sigma' \sqsubseteq_M \tilde{v}_1 :: \tilde{v}_2, \tilde{\sigma}', \Phi \wedge \phi$. From this, we can conclude that $\text{tail } e, \sigma \downarrow v_2, \sigma'$.

Case SE-APP

Provided that $e, \sigma \sqsubseteq_M \tilde{e}_1 \tilde{e}_2, \tilde{\sigma}, \Phi$ and $\tilde{e}_1 \tilde{e}_2, \tilde{\sigma} \downarrow \tilde{v}_1, \tilde{\sigma}''', \phi_1 \wedge \phi_2 \wedge \phi_3$, we know that $e = e_1 e_2$. We obtain from the induction hypothesis that $e_1, \sigma \downarrow \lambda x : \tau. e_1', \sigma'$ with $\lambda x : \tau. e_1', \sigma' \sqsubseteq_M \lambda x : \tau. \tilde{e}_1', \tilde{\sigma}', \Phi \wedge \phi_1$.

Then by a second application of the induction hypothesis, we obtain that $e_2, \sigma' \downarrow v_2, \sigma''$ with $v_2, \sigma'' \sqsubseteq_M \tilde{v}_2, \tilde{\sigma}'', \Phi \wedge \phi_1 \wedge \phi_2$.

A third and final application of the induction hypothesis gives us that $e_1'[x \mapsto v_2], \sigma'' \downarrow v_1, \sigma'''$ with $v_1, \sigma''' \sqsubseteq_M \tilde{v}_1, \tilde{\sigma}''', \Phi \wedge \phi_1 \wedge \phi_2 \wedge \phi_3$. From this, we can conclude that $e_1 e_2, \sigma \downarrow v_1, \sigma'''$.

Case SE-IF

Provided that $e, \sigma \sqsubseteq_M$ **if** $\tilde{e}_1$ **then** $\tilde{e}_2$ **else** $\tilde{e}_3, \tilde{\sigma}, \Phi$ and
 if $\tilde{e}_1$ **then** $\tilde{e}_2$ **else** $\tilde{e}_3, \tilde{\sigma} \downarrow \overline{\tilde{v}_2, \tilde{\sigma}'', \phi_1 \wedge \phi_2 \wedge \tilde{v}_1 \cup \tilde{v}_3, \tilde{\sigma}''', \phi_1 \wedge \phi_3 \wedge \neg \tilde{v}_1}$, we

know that $e = \text{if } e_1 \text{ then } e_2 \text{ else } e_3$. We obtain from the induction hypothesis that $e_1, \sigma \downarrow v_1, \sigma'$ with $v_1, \sigma' \sqsubseteq_M \tilde{v}_1, \tilde{\sigma}', \Phi \wedge \phi_1$. At this point, we have two potential branches.

In case of v_1 , we obtain from application of the induction hypothesis that $e_2, \sigma' \downarrow v_2, \sigma''$ with $v_2, \sigma'' \sqsubseteq_M \tilde{v}_2, \tilde{\sigma}'', \Phi \wedge \phi_1 \wedge \phi_2$. From this, we can conclude that **if** e_1 **then** e_2 **else** $e_3, \sigma \downarrow v_2, \sigma''$ with $v_2, \sigma'' \sqsubseteq_M \tilde{v}_2, \tilde{\sigma}'', \Phi \wedge \phi_1 \wedge \phi_2$.

And in case of $\neg v_1$, we obtain from application of the induction hypothesis that $e_3, \sigma' \downarrow v_3, \sigma''$ with $v_3, \sigma'' \sqsubseteq_M \tilde{v}_3, \tilde{\sigma}'', \Phi \wedge \phi_1 \wedge \phi_3$. From this, we can conclude that **if** e_1 **then** e_2 **else** $e_3, \sigma \downarrow v_3, \sigma''$ with $v_3, \sigma'' \sqsubseteq_M \tilde{v}_3, \tilde{\sigma}'', \Phi \wedge \phi_1 \wedge \phi_3$.

Case SE-REF

Provided that $e, \sigma \sqsubseteq_M \text{ref } \tilde{e}, \tilde{\sigma}, \Phi$ and $\text{ref } \tilde{e}, \tilde{\sigma} \downarrow l, \tilde{\sigma}'[l \mapsto \tilde{v}], \phi$, we know that $e = \text{ref } e$. We obtain from the induction hypothesis that $e, \sigma \downarrow v_1, \sigma'$ with $v_1, \sigma' \sqsubseteq_M \tilde{v}_1, \tilde{\sigma}', \Phi \wedge \phi$. From this, we can conclude that $\text{ref } e, \sigma \downarrow l, \sigma'[l \mapsto v]$ with $l, \sigma'[l \mapsto v] \sqsubseteq_M l, \tilde{\sigma}'[l \mapsto \tilde{v}], \Phi \wedge \phi$.

Case SE-DEREF

Provided that $e, \sigma \sqsubseteq_M !\tilde{e}, \tilde{\sigma}, \Phi$ and $!\tilde{e}, \tilde{\sigma} \downarrow \tilde{\sigma}'(l), \tilde{\sigma}', \phi$, we know that $e = !e$. We obtain from the induction hypothesis that $e, \sigma \downarrow l, \sigma'$ with $l, \sigma' \sqsubseteq_M l, \tilde{\sigma}', \Phi \wedge \phi$. From this, we can conclude that $!e, \sigma \downarrow \sigma'(l), \sigma'$ with $\sigma'(l), \sigma' \sqsubseteq_M \tilde{\sigma}'(l), \tilde{\sigma}', \Phi \wedge \phi$.

Case SE-ASSIGN

Provided that $e, \sigma \sqsubseteq_M \tilde{e}_1 := \tilde{e}_2, \tilde{\sigma}, \Phi$ and $\tilde{e}_1 := \tilde{e}_2, \tilde{\sigma} \downarrow \langle \rangle, \tilde{\sigma}''[l \mapsto \tilde{v}_2], \phi_1 \wedge \phi_2$, we know that $e = e_1 := e_2$. We obtain from the induction hypothesis that $e_1, \sigma \downarrow l, \sigma'$ with $l, \sigma' \sqsubseteq_M l, \tilde{\sigma}', \Phi \wedge \phi_1$. Then by a second application of the induction hypothesis, we obtain that $e_2, \sigma' \downarrow v_2, \sigma''$ with $v_2, \sigma'' \sqsubseteq_M \tilde{v}_2, \tilde{\sigma}'', \Phi \wedge \phi_1 \wedge \phi_2$. From this, we can conclude that $e_1 := e_2, \sigma \downarrow \langle \rangle, \sigma''[l \mapsto v_2]$ with $\langle \rangle, \sigma''[l \mapsto v_2] \sqsubseteq_M \langle \rangle, \tilde{\sigma}''[l \mapsto \tilde{v}_2], \Phi \wedge \phi_1 \wedge \phi_2$.

Case SE-EDIT

Provided that $e, \sigma \sqsubseteq_M \Box \tilde{e}, \tilde{\sigma}, \Phi$ and $\Box \tilde{e}, \tilde{\sigma} \downarrow \Box \tilde{v}, \tilde{\sigma}', \phi$, we know that $e = \Box e$. We obtain from the induction hypothesis that $e, \sigma \downarrow v, \sigma'$ with $v, \sigma' \sqsubseteq_M \tilde{v}, \tilde{\sigma}', \Phi \wedge \phi$. From this, we can conclude that $\Box e, \sigma \downarrow \Box v, \sigma'$ with $\Box v, \sigma' \sqsubseteq_M \Box \tilde{v}, \tilde{\sigma}', \Phi \wedge \phi$.

Case SE-UPDATE

Provided that $e, \sigma \sqsubseteq_M \blacksquare \tilde{e}, \tilde{\sigma}, \Phi$ and $\blacksquare \tilde{e}, \tilde{\sigma} \downarrow \blacksquare l, \tilde{\sigma}', \phi$, we know that $e = \blacksquare e$. We obtain from the induction hypothesis that $e, \sigma \downarrow l, \sigma'$ with

$l, \sigma' \sqsubseteq_M l, \tilde{\sigma}', \Phi \wedge \phi$. From this, we can conclude that $\blacksquare e, \sigma \downarrow \blacksquare l, \sigma'$ with $\blacksquare l, \sigma' \sqsubseteq_M \blacksquare l, \tilde{\sigma}', \Phi \wedge \phi$.

Case SE-THEN

Provided that $e, \sigma \sqsubseteq_M \tilde{e}_1 \triangleright \tilde{e}_2, \tilde{\sigma}, \Phi$ and $\tilde{e}_1 \triangleright \tilde{e}_2, \tilde{\sigma} \Downarrow \tilde{t}_1 \triangleright \tilde{e}_2, \tilde{\sigma}', \phi$, we know that $e = e_1 \triangleright e_2$. We obtain from the induction hypothesis that $e_1, \sigma \downarrow t_1, \sigma'$ with $t_1, \sigma' \sqsubseteq_M \tilde{t}_1, \tilde{\sigma}', \Phi \wedge \phi$. From this, we can conclude that $e_1 \triangleright e_2, \sigma \downarrow t_1 \triangleright e_2, \sigma'$ with $t_1 \triangleright e_2, \sigma' \sqsubseteq_M \tilde{t}_1 \triangleright e_2, \tilde{\sigma}', \Phi \wedge \phi$.

Case SE-NEXT

Provided that $e, \sigma \sqsubseteq_M \tilde{e}_1 \triangleright \tilde{e}_2, \tilde{\sigma}, \Phi$ and $\tilde{e}_1 \triangleright \tilde{e}_2, \tilde{\sigma} \Downarrow \tilde{t}_1 \triangleright \tilde{e}_2, \tilde{\sigma}', \phi$, we know that $e = e_1 \triangleright e_2$. We obtain from the induction hypothesis that $e_1, \sigma \downarrow t_1, \sigma'$ with $t_1, \sigma' \sqsubseteq_M \tilde{t}_1, \tilde{\sigma}', \Phi \wedge \phi$. From this, we can conclude that $e_1 \triangleright e_2, \sigma \downarrow t_1 \triangleright e_2, \sigma'$ with $t_1 \triangleright e_2, \sigma' \sqsubseteq_M \tilde{t}_1 \triangleright e_2, \tilde{\sigma}', \Phi \wedge \phi$.

Case SE-OR

Provided that $e, \sigma \sqsubseteq_M \tilde{e}_1 \blacklozenge \tilde{e}_2, \tilde{\sigma}, \Phi$ and $\tilde{e}_1 \blacklozenge \tilde{e}_2, \tilde{\sigma} \Downarrow \tilde{v}_1 \blacklozenge \tilde{v}_2, \tilde{\sigma}'', \phi_1 \wedge \phi_2$, we know that $e = e_1 \blacklozenge e_2$. We obtain from the induction hypothesis that $e_1, \sigma \downarrow v_1, \sigma'$ with $v_1, \sigma' \sqsubseteq_M \tilde{v}_1, \tilde{\sigma}', \Phi \wedge \phi_1$. Then by a second application of the induction hypothesis, we obtain that $e_2, \sigma' \downarrow v_2, \sigma''$ with $v_2, \sigma'' \sqsubseteq_M \tilde{v}_2, \tilde{\sigma}'', \Phi \wedge \phi_1 \wedge \phi_2$. From this, we can conclude that $e_1 \blacklozenge e_2, \sigma \downarrow v_1 \blacklozenge v_2, \sigma''$ with $v_1 \blacklozenge v_2, \sigma'' \sqsubseteq_M \tilde{v}_1 \blacklozenge \tilde{v}_2, \tilde{\sigma}'', \Phi \wedge \phi_1 \wedge \phi_2$. $\square$

E.2.7 $\mathcal{V}$ preserves consistency

Lemma E.2.5 ($\mathcal{V}$ preserves consistency)

For all concrete tasks t , concrete states σ , symbolic tasks $\tilde{t}$, symbolic states $\tilde{\sigma}$, path conditions Φ and mappings $M = [s_1 \mapsto c_1 \cdots s_n \mapsto c_n]$, if $t, \sigma \sqsubseteq_M \tilde{t}, \tilde{\sigma}, \Phi$ and $\mathcal{V}(t, \sigma) = v$ and $\mathcal{V}(\tilde{t}, \tilde{\sigma})$, then also $v, \sigma \sqsubseteq_M \tilde{v}, \tilde{\sigma}, \Phi$

Proof: We prove Lemma E.2.5 by induction over $\tilde{t}$.

Case $\tilde{t} = \square s$

If we have $t, \sigma \sqsubseteq_M \square s, \tilde{\sigma}, \Phi$, then by definition of consistency t must be $\square c$ for some concrete value of the same type as s .

Then by definition of $\mathcal{V}$, we have $\mathcal{V}(\square c, \sigma) = c$ and $\mathcal{V}(\square s, \tilde{\sigma}) = s$. Since we have $M(\square s) = \square c$ from the premise, we know that $Ms = c$, since mapping propagates. Therefore $c, \sigma \sqsubseteq_M s, \tilde{\sigma}, \Phi$.

Case $\tilde{t} = \boxtimes \beta$

If we have $t, \sigma \sqsubseteq_M \boxtimes \beta, \tilde{\sigma}, \Phi$, then t is also $\boxtimes \beta$ by definition of consistency.

By definition of $\mathcal{V}$, $\mathcal{V}(\boxtimes \beta, \sigma) = \perp$ and $\mathcal{V}(\boxtimes \beta, \tilde{\sigma}) = \perp$, so this case holds trivially.

Case $\tilde{t} = \blacksquare l$

If we have $t, \sigma \sqsubseteq_M \blacksquare l, \tilde{\sigma}, \Phi$, then t is also $\blacksquare l$ by definition of consistency.

By definition of $\mathcal{V}$, $\mathcal{V}(\blacksquare l, \sigma) = \sigma(l)$ and $\mathcal{V}(\blacksquare l, \tilde{\sigma}) = \tilde{\sigma}(l)$.

We now need to show that $M(\tilde{\sigma}(l)) = \sigma(l)$. From the premise we know that $M\tilde{\sigma} = \sigma$, from which this immediately follows.

Case $\tilde{t} = \downarrow$

If we have $t, \sigma \sqsubseteq_M \downarrow, \tilde{\sigma}, \Phi$, then t is also $\downarrow$ by definition of consistency.

By definition of $\mathcal{V}$, $\mathcal{V}(\downarrow, \sigma) = \perp$ and $\mathcal{V}(\downarrow, \tilde{\sigma}) = \perp$, so we know that this case holds trivially.

Case $\tilde{t} = \tilde{t}_1 \blacktriangleright \tilde{e}_2$

If we have $t, \sigma \sqsubseteq_M \tilde{t}_1 \blacktriangleright \tilde{e}_2, \tilde{\sigma}, \Phi$, then t is $t_1 \blacktriangleright e_2$ by definition of consistency.

By definition of $\mathcal{V}$, $\mathcal{V}(t_1 \blacktriangleright e_2, \sigma) = \perp$ and $\mathcal{V}(\tilde{t}_1 \blacktriangleright \tilde{e}_2, \tilde{\sigma}) = \perp$, so we know that this case holds trivially.

Case $\tilde{t} = \tilde{t}_1 \triangleright \tilde{e}_2$

If we have $t, \sigma \sqsubseteq_M \tilde{t}_1 \triangleright \tilde{e}_2, \tilde{\sigma}, \Phi$, then t is $t_1 \triangleright e_2$ by definition of consistency.

By definition of $\mathcal{V}$, $\mathcal{V}(t_1 \triangleright e_2, \sigma) = \sigma(l)$ and $\mathcal{V}(\tilde{t}_1 \triangleright \tilde{e}_2, \tilde{\sigma}) = \perp$, so we know that this case holds trivially.

Case $\tilde{t} = \tilde{t}_1 \bowtie \tilde{t}_2$

If we have $t, \sigma \sqsubseteq_M \tilde{t}_1 \bowtie \tilde{t}_2, \tilde{\sigma}, \Phi$, then t is also $t_1 \bowtie t_2$ by definition of consistency.

By definition of $\mathcal{V}$, we can find ourselves in one of two cases.

If $\mathcal{V}(\tilde{t}_1, \sigma) = \tilde{v}_1$ and $\mathcal{V}(\tilde{t}_2, \sigma) = \tilde{v}_2$, then $\mathcal{V}(t_1 \bowtie t_2, \sigma) = \langle v_1, v_2 \rangle$ and $\mathcal{V}(\tilde{t}_1 \bowtie \tilde{t}_2, \tilde{\sigma}) = \langle \tilde{v}_1, \tilde{v}_2 \rangle$. This case follows from the induction hypothesis.

Otherwise, if either one of the two branches returns $\perp$, we have that $\mathcal{V}(t_1 \bowtie t_2, \sigma) = \perp$ and $\mathcal{V}(\tilde{t}_1 \bowtie \tilde{t}_2, \tilde{\sigma}) = \perp$, so we know that this case holds trivially.

Case $\tilde{t} = \tilde{t}_1 \blacklozenge \tilde{t}_2$

If we have $t, \sigma \sqsubseteq_M \tilde{t}_1 \blacklozenge \tilde{t}_2, \tilde{\sigma}, \Phi$, then t is also $t_1 \blacklozenge t_2$ by definition of consistency.

By definition of $\mathcal{V}$, we find ourselves in one of three cases.

If $\mathcal{V}(\tilde{t}_1, \tilde{\sigma}) = \tilde{v}_1$, then $\mathcal{V}(\tilde{t}_1 \blacklozenge \tilde{t}_2, \tilde{\sigma}) = \tilde{v}_1$ and $\mathcal{V}(t_1 \blacklozenge t_2, \sigma) = v_1$. This case follows from the induction hypothesis.

Otherwise, if $\mathcal{V}(\tilde{t}_2, \tilde{\sigma}) = \tilde{v}_2$, then $\mathcal{V}(\tilde{t}_1 \diamond \tilde{t}_2, \tilde{\sigma}) = \tilde{v}_2$ and $\mathcal{V}(t_1 \diamond t_2, \sigma) = v_2$. This case follows from the induction hypothesis.

Otherwise, if either one of the two branches returns $\perp$, we have that $\mathcal{V}(t_1 \diamond t_2, \sigma) = \perp$ and $\mathcal{V}(\tilde{t}_1 \diamond \tilde{t}_2, \tilde{\sigma}) = \perp$, so we know that this case holds trivially.

Case $\tilde{t} = \tilde{t}_1 \diamond \tilde{t}_2$

If we have $t, \sigma \hookrightarrow_M \tilde{t}_1 \diamond \tilde{t}_2, \tilde{\sigma}, \Phi$, then t is $t_1 \diamond t_2$ by definition of consistency.

By definition of $\mathcal{V}$, $\mathcal{V}(t_1 \diamond t_2, \sigma) = \perp$ and $\mathcal{V}(\tilde{t}_1 \diamond \tilde{t}_2, \tilde{\sigma}) = \perp$, so we know that this case holds trivially.

□

References

- Aalst, W. M. P. v. d. (1998). The application of petri nets to workflow management. *Journal of Circuits, Systems, and Computers*, 8(1), 21–66.
- Aalst, W. M. P. v. d. (2011). *Process mining - discovery, conformance and enhancement of business processes*. Springer.
- Aalst, W. M. P. v. d., & ter Hofstede, A. H. M. (2005). YAWL: yet another workflow language. *Information Systems*, 30(4), 245–275.
- Aalst, W. M. P. v. d., ter Hofstede, A. H. M., Kiepuszewski, B., & Barros, A. P. (2003). Workflow patterns. *Distributed and Parallel Databases*, 14(1), 5–51.
- Aleven, V., McLaren, B. M., Sewall, J., & Koedinger, K. R. (2009). A new paradigm for intelligent tutoring systems: Example-tracing tutors. *I. J. Artificial Intelligence in Education*, 19(2), 105–154.
- Apfelmus, H. (2019). *reactive-banana*. <https://hackage.haskell.org/package/reactive-banana>. (Accessed 13-February-2019)
- Barker, P., & Banerji, A. (1995). Designing electronic performance support systems. *Innovations in Education and Training International*, 32(1), 4–12.
- Basu, A., & Blanning, R. W. (2000). A formal approach to workflow analysis. *Information Systems Research*, 11(1), 17–36.
- Berner, E. S., & La Lande, T. J. (2007). Overview of clinical decision support systems. In *Clinical decision support systems* (pp. 3–22). Springer.
- Berry, G., & Gonthier, G. (1992). The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2), 87–152.
- Berry, G., Nicolas, C., & Serrano, M. (2011). HipHop: A synchronous reactive extension for Hop. In *Proceedings of the 1st ACM SIGPLAN international workshop on programming language and systems technologies for internet clients* (pp. 49–56).
- Berry, G., & Serrano, M. (2014). Hop and HipHop : Multitier web orchestration. In *International conference on distributed computing and internet technology* (pp. 1–13).
- Boussinot, F., & De Simone, R. (1991). The Esterel language. *Proceedings of the IEEE*, 79(9), 1293–1304.
- Boyer, R. S., Elspas, B., & Levitt, K. N. (1975). SELECT - a formal system for

- testing and debugging programs by symbolic execution. In *Proceedings of the international conference on reliable software* (pp. 234–245). ACM.
- Bucur, S., Kinder, J., & Candea, G. (2014). Prototyping symbolic execution engines for interpreted languages. In *Architectural support for programming languages and operating systems* (pp. 239–254).
- Bylander, T. (1994). The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, 69(1-2), 165–204.
- Cadar, C., Dunbar, D., & Engler, D. R. (2008). KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings eight USENIX symposium on operating systems design and implementation* (pp. 209–224). USENIX Association.
- Chang, S., Knauth, A., & Torlak, E. (2018). Symbolic types for lenient symbolic execution. *Proceedings of the ACM on Programming Languages*, POPL'18, 2, 40:1–40:29.
- Claessen, K., & Hughes, J. (2000). QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the fifth ACM SIGPLAN international conference on functional programming, ICFP'00* (pp. 268–279).
- Cooper, G., & Krishnamurthi, S. (2004). *FrTime: Functional reactive programming in plt scheme* (Tech. Rep. No. CS-03-20). Rhode Island: Department of Computer Science, Brown University.
- Downey, A. B. (2008). *The little book of semaphores*. Green Tea Press.
- Elliott, C., & Hudak, P. (1997). Functional reactive animation. In *Proceedings of the second ACM SIGPLAN international conference on functional programming, ICFP'97* (pp. 263–273).
- Fikes, R., & Nilsson, N. J. (1971). STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3-4), 189–208.
- Findler, R. B., & Felleisen, M. (2002). Contracts for higher-order functions. In *Proceedings of the seventh ACM SIGPLAN international conference on functional programming, ICFP'02* (pp. 48–59).
- Galagan, N. (1979). Problem description language SITPLAN. *Cybernetics and Systems Analysis*, 15(2), 255–266.
- Gerdes, A., Heeren, B., & Jeuring, J. (2012). Teachers and students in charge - using annotated model solutions in a functional programming tutor. In *21st century learning for 21st century skills - 7th european conference of technology enhanced learning, EC-TEL'12* (pp. 383–388).
- Gerdes, A., Heeren, B., Jeuring, J., & van Binsbergen, L. T. (2017). Ask-Elle: an adaptable programming tutor for Haskell giving automated feedback. *I. J. Artificial Intelligence in Education*, 27(1), 65–100.

- Gerdes, A., Jeuring, J., & Heeren, B. (2012). An interactive functional programming tutor. In *Proceedings of iticse 2012: the 17th annual conference on innovation and technology in computer science education* (pp. 250–255). ACM.
- Giantsios, A., Papaspyrou, N., & Sagonas, K. (2017). Concolic testing for functional languages. *Science of Computer Programming*, 147, 109–134.
- Hallahan, W. T., Xue, A., Bland, M. T., Jhala, R., & Piskac, R. (2019). Lazy counterfactual symbolic execution. In *Proceedings of the 40th ACM SIGPLAN conference on programming language design and implementation, PLDI'19* (pp. 411–424).
- Hallahan, W. T., Xue, A., & Piskac, R. (2017). Building a symbolic execution engine for Haskell. In *Proceedings of TAPAS 17*.
- Heeren, B., & Jeuring, J. (2014). Feedback services for stepwise exercises. *Science of Computer Programming*, 88, 110–129.
- Heeren, B., Jeuring, J., & Gerdes, A. (2010). Specifying rewrite strategies for interactive exercises. *Mathematics in Computer Science*, 3(3), 349–370.
- Hewitt, C. (1969). PLANNER: A language for proving theorems in robots. In *Proceedings of the 1st international joint conference on artificial intelligence* (pp. 295–302).
- Hoare, C. A. R. (1985). *Communicating sequential processes*. Prentice Hall.
- Hoare, T. (1969). An axiomatic basis for computer programming. *CACM: Communications of the ACM*, 12.
- Jaffar, J., Murali, V., Navas, J. A., & Santosa, A. E. (2012). TRACER: A symbolic execution tool for verification. In P. Madhusudan & S. A. Seshia (Eds.), *Computer aided verification, CAV'12* (pp. 758–766). Springer-Verlag.
- Jansen, J., & Bolderheij, F. (2018). Dynamic resource and task management. In *Nl arms netherlands annual review of military studies 2018* (pp. 91–105). Springer.
- Jaskelioff, M., Ghani, N., & Hutton, G. (2011). Modularity and implementation of mathematical operational semantics. *Electronic Notes on Theoretical Computer Science*, 229(5), 75–95.
- Jeuring, J., Grosfeld, F., Heeren, B., Hulsbergen, M., IJntema, R., Jonker, V., . . . Wolters, M. (2015). Communicate! A serious game for communication skills—. In *Proceedings of EC-TEL 2015: Design for teaching and learning in a networked world* (Vol. 9307, pp. 513–517). Springer.
- Junghanns, A., & Schaeffer, J. (1997). Sokoban: A challenging single-agent search problem. In *In ijcai workshop on using games as an experimental testbed for ai reasearch*.
- Kaelbling, L. P., Littman, M. L., & Cassandra, A. R. (1998). Planning and acting in partially observable stochastic domains. *Artificial Intelligence*,

- 101(1-2), 99-134.
- Keuning, H., Jeurig, J., & Heeren, B. (2019). A systematic literature review of automated feedback generation for programming exercises. *TOCE*, 19(1), 3:1–3:43.
- King, J. C. (1975). A new approach to program testing. *SIGPLAN Notices*, 10(6), 228–233.
- Klinik, M., Jansen, J. M., & Plasmeijer, R. (2017). The sky is the limit: Analysing resource consumption over time using skylines. In *Proceedings of the 29th symposium on implementation and application of functional programming languages, IFL'17*. ACM.
- Kool, B. (2017). *Integrated mission management voor C2-ondersteuning*. Bachelor's Thesis. (Dutch Defence Academy, Den Helder, The Netherlands)
- Koopman, P., Lubbers, M., & Plasmeijer, R. (2018). A task-based DSL for microcomputers. In *Proceedings of the real world domain specific languages workshop, RWDSL@CGO'18* (pp. 4:1–4:11). ACM.
- Koopman, P. W. M., Plasmeijer, R., & Achten, P. (2008). An executable and testable semantics for itasks. In *Proceedings of the 20th symposium on implementation and application of functional programming languages, IFL'08*.
- Kovacs, D. L. (2011). *BNF definition of PDDL 3.1*.
- Kovacs, D. L. (2012). A multi-agent extension of PDDL3. *WS-IPC 2012*, 19.
- Kulik, J. A., & Fletcher, J. (2016). Effectiveness of intelligent tutoring systems: a meta-analytic review. *Review of Educational Research*, 86(1), 42–78.
- Lavelle, S. (2016). *PuzzleScript*. <https://www.puzzlescript.net>. (Accessed 12-February-2019)
- Lijnse, B., Jansen, J. M., & Plasmeijer, R. (2012). Incidone: A task-oriented incident coordination tool. In *Proceedings of ISCRAM*.
- Lim, C., & Harrell, D. F. (2014). An approach to general videogame evaluation and automatic generation using a description language. In *Proceedings of IEEE CIG 2014: Conference on computational intelligence and games* (pp. 1–8).
- Luger, G. F. (2005). *Artificial intelligence: structures and strategies for complex problem solving*. Pearson education.
- Marlow, S. (2010). Haskell 2010 language report [Computer software manual]. <http://www.haskell.org/>.
- McDermott, D., Ghallab, M., Howe, A., Knoblock, C., Ram, A., Veloso, M., ... Wilkins, D. (1998). Pddl-the planning domain definition language. *AIPS-98 planning committee*, 3, 14.
- Meyer, B. (1992). Applying "design by contract". *IEEE Computer*, 25(10), 40–51.
- Meyerovich, L. A., Guha, A., Baskin, J. P., Cooper, G. H., Greenberg, M.,

- Bromfield, A., & Krishnamurthi, S. (2009). Flapjax: a programming language for ajax applications. In *Proceedings of the 24th annual ACM SIGPLAN conference on object-oriented programming, systems, languages, and applications, OOPSLA'09* (pp. 1–20).
- Milner, R. (1989). *Communication and concurrency*. Prentice Hall.
- Murray, T. (2003). An overview of intelligent tutoring system authoring tools: Updated analysis of the state of the art. In *Authoring tools for advanced technology learning environments* (pp. 491–544). Springer.
- Nanevski, A., Morrisett, G., & Birkedal, L. (2006). Polymorphism and separation in Hoare type theory. *ACM SIGPLAN Notices*, 41(9), 62–73.
- Nanevski, A., Morrisett, G., Shinnar, A., Govereau, P., & Birkedal, L. (2008). Ynot: dependent types for imperative programs. In *Proceeding of the 13th ACM SIGPLAN international conference on functional programming, ICFP'08* (pp. 229–240).
- Nguyen, P. C., Tobin-Hochstadt, S., & Horn, D. V. (2017). Higher order symbolic execution for contract verification and refutation. *Journal of Functional Programming*, 27.
- Nielson, H. R., & Nielson, F. (1992). *Semantics with applications - a formal introduction*. Wiley.
- OASIS. (2019). *Web services business process execution language*. (Accessed 12-February-2019)
- Oortgiese, A., van Groningen, J. H. G., Achten, P., & Plasmeijer, R. (2017). A distributed dynamic architecture for task oriented programming. In *Proceedings of the 29th symposium on implementation and application of functional programming languages, IFL'17* (pp. 7:1–7:12).
- Paquette, L., Lebeau, J., Beaulieu, G., & Mayers, A. (2012). Automating next-step hints generation using ASTUS. In S. A. Cerri, W. J. Clancey, G. Papadourakis, & K. Panourgia (Eds.), *Intelligent tutoring systems - 11th international conference* (pp. 201–211). Springer.
- Pearl, J. (1989). *Probabilistic reasoning in intelligent systems - networks of plausible inference*. Morgan Kaufmann.
- Peyton Jones, S. (2001). Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In *Engineering theories of software construction, Marktoberdorf summer school 2000*.
- Pierce, B. C. (2002). *Types and programming languages*. MIT Press.
- Plasmeijer, R., Achten, P., Koopman, P. W. M., Lijnse, B., van Noort, T., & van Groningen, J. H. G. (2011). iTasks for a change: type-safe run-time change in dynamically evolving workflows. In *Proceedings of the 2011 ACM SIGPLAN workshop on partial evaluation and program manipulation, PEPM'11* (pp. 151–160). ACM.

- Plasmeijer, R., Lijnse, B., Michels, S., Achten, P., & Koopman, P. W. M. (2012). Task-oriented programming in a pure functional language. In *Principles and practice of declarative programming, PPDP'12* (pp. 195–206).
- Plasmeijer, R., van Eekelen, M., & van Groningen, J. (2002). *Clean language report version 2.1*.
- Power, D. J. (2002). *Decision support systems: concepts and resources for managers*. Greenwood Publishing Group.
- Reinefeld, A. (1993). Complete solution of the eight-puzzle and the benefit of node ordering in IDA. In *Proceedings of the 13th international joint conference on artificial intelligence* (pp. 248–253).
- Rosa, D. d. I., Mayol, F., Díaz-Pereira, E., Fernandez, M., & Rosa Jr., D. d. I. (2004). A land evaluation decision support system (microleis DSS) for agricultural soil protection: With special reference to the mediterranean region. *Environmental Modelling and Software*, 19(10), 929–942.
- Russell, S. J., & Norvig, P. (2010). *Artificial intelligence - A modern approach* (3. internat. ed.). Pearson Education.
- Schaik, P. V., Pearson, R., & Barker, P. (2002). Designing electronic performance support systems to facilitate learning. *Innovations in Education and Teaching International*, 39(4), 289–306.
- Sharda, R., Barr, S. H., & McDonnell, J. C. (1988). Decision support system effectiveness: a review and an empirical test. *Management science*, 34(2), 139–159.
- Shim, J. P., Warkentin, M., Courtney, J. F., Power, D. J., Sharda, R., & Carlsson, C. (2002). Past, present, and future of decision support technology. *Decision Support Systems*, 33(2), 111–126.
- Sottolare, R., Graesser, A., Hu, X., & Brawner, K. (2015). *Design recommendations for intelligent tutoring systems: Authoring tools and expert modeling techniques*. Robert Sottolare.
- Steenvoorden, T., Naus, N., & Klinik, M. (2019). Tophat: A formal foundation for task-oriented programming. In *Proceedings of the 21st international symposium on principles and practice of programming languages, PPDP'19* (pp. 17:1–17:13).
- Stutterheim, J. (2017). *A cocktail of tools* (Unpublished doctoral dissertation). Radboud University, Nijmegen, The Netherlands.
- Stutterheim, J., Achten, P., & Plasmeijer, R. (2015). Static and Dynamic Visualisations of Monadic Programs. In *Implementation and application of functional languages, IFL'15* (pp. 1–13).
- Stutterheim, J., Achten, P., & Plasmeijer, R. (2016). C2 demo.
- Stutterheim, J., Achten, P., & Plasmeijer, R. (2017). Maintaining separation of concerns through task oriented software development. In *18th*

- international symposium on trends in functional programming, TFP'17.*
- Stutterheim, J., Plasmeijer, R., & Achten, P. (2014). Tonic: An infrastructure to graphically represent the definition and behaviour of tasks. In *Trends in functional programming - 15th international symposium, TFP'14* (pp. 122–141).
- Swamy, N., Weinberger, J., Schlesinger, C., Chen, J., & Livshits, B. (2013). Verifying higher-order programs with the dijkstra monad. In *ACM SIGPLAN conference on programming language design and implementation, PLDI '13* (pp. 387–398).
- Swierstra, W. (2008). Data types à la carte. *Journal of Functional Programming*, 18(4), 423–436.
- Swierstra, W. (2009). A hoare logic for the state monad. In *22nd international conference on theorem proving in higher order logics, TPHOLs'09* (pp. 440–451). Springer.
- Turban, E. (1988). *Decision support and expert systems: Managerial perspectives*. Macmillan.
- Visser, E., Benaissa, Z., & Tolmach, A. P. (1998). Building program optimizers with rewriting strategies. In *Proceedings of the third ACM SIGPLAN international conference on functional programming (ICFP '98)* (pp. 13–26).
- Winskel, G. (1993). *The formal semantics of programming languages - an introduction*. MIT Press.
- Younes, H. L., & Littman, M. L. (2004). PPDDL1. 0: The language for the probabilistic part of ipc-4. In *Proc. international planning competition*.

Titles in the IPA Dissertation Series since 2017

M.J. Steindorfer. *Efficient Immutable Collections*. Faculty of Science, UvA. 2017-01

W. Ahmad. *Green Computing: Efficient Energy Management of Multiprocessor Streaming Applications via Model Checking*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2017-02

D. Guck. *Reliable Systems – Fault tree analysis via Markov reward automata*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2017-03

H.L. Salunkhe. *Modeling and Buffer Analysis of Real-time Streaming Radio Applications Scheduled on Heterogeneous Multiprocessors*. Faculty of Mathematics and Computer Science, TU/e. 2017-04

A. Krasnova. *Smart invaders of private matters: Privacy of communication on the Internet and in the Internet of Things (IoT)*. Faculty of Science, Mathematics and Computer Science, RU. 2017-05

A.D. Mehrabi. *Data Structures for Analyzing Geometric Data*. Faculty

of Mathematics and Computer Science, TU/e. 2017-06

D. Landman. *Reverse Engineering Source Code: Empirical Studies of Limitations and Opportunities*. Faculty of Science, UvA. 2017-07

W. Lueks. *Security and Privacy via Cryptography – Having your cake and eating it too*. Faculty of Science, Mathematics and Computer Science, RU. 2017-08

A.M. Şutîi. *Modularity and Reuse of Domain-Specific Languages: an exploration with MetaMod*. Faculty of Mathematics and Computer Science, TU/e. 2017-09

U. Tikhonova. *Engineering the Dynamic Semantics of Domain Specific Languages*. Faculty of Mathematics and Computer Science, TU/e. 2017-10

Q.W. Bouts. *Geographic Graph Construction and Visualization*. Faculty of Mathematics and Computer Science, TU/e. 2017-11

A. Amighi. *Specification and Verification of Synchronisation Classes in Java:*

A Practical Approach. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-01

S. Darabi. *Verification of Program Parallelization*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-02

J.R. Salamanca Tellez. *Coequations and Eilenberg-type Correspondences*. Faculty of Science, Mathematics and Computer Science, RU. 2018-03

P. Fiterău-Broștean. *Active Model Learning for the Analysis of Network Protocols*. Faculty of Science, Mathematics and Computer Science, RU. 2018-04

D. Zhang. *From Concurrent State Machines to Reliable Multi-threaded Java Code*. Faculty of Mathematics and Computer Science, TU/e. 2018-05

H. Basold. *Mixed Inductive-Coinductive Reasoning Types, Programs and Logic*. Faculty of Science, Mathematics and Computer Science, RU. 2018-06

A. Lele. *Response Modeling: Model Refinements for Timing Analysis of Runtime Scheduling in Real-time Streaming Systems*. Faculty of Mathematics and Computer Science, TU/e. 2018-07

N. Bezirgiannis. *Abstract Behavioral Specification: unifying modeling and programming*. Faculty of Mathematics and Natural Sciences, UL. 2018-08

M.P. Konzack. *Trajectory Analysis: Bridging Algorithms and Visualization*. Faculty of Mathematics and Computer Science, TU/e. 2018-09

E.J.J. Ruijters. *Zen and the art of railway maintenance: Analysis and optimization of maintenance via fault trees and statistical model checking*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-10

F. Yang. *A Theory of Executability: with a Focus on the Expressivity of Process Calculi*. Faculty of Mathematics and Computer Science, TU/e. 2018-11

L. Swartjes. *Model-based design of baggage handling systems*. Faculty of Mechanical Engineering, TU/e. 2018-12

T.A.E. Ophelders. *Continuous Similarity Measures for Curves and Surfaces*. Faculty of Mathematics and Computer Science, TU/e. 2018-13

M. Talebi. *Scalable Performance Analysis of Wireless Sensor Network*. Faculty of Mathematics and Computer Science, TU/e. 2018-14

R. Kumar. *Truth or Dare: Quantitative security analysis using attack trees*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-15

M.M. Beller. *An Empirical Evaluation of Feedback-Driven Software Development*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2018-16

- M. Mehr.** *Faster Algorithms for Geometric Clustering and Competitive Facility-Location Problems.* Faculty of Mathematics and Computer Science, TU/e. 2018-17
- M. Alizadeh.** *Auditing of User Behavior: Identification, Analysis and Understanding of Deviations.* Faculty of Mathematics and Computer Science, TU/e. 2018-18
- P.A. Inostroza Valdera.** *Structuring Languages as Object-Oriented Libraries.* Faculty of Science, UvA. 2018-19
- M. Gerhold.** *Choice and Chance - Model-Based Testing of Stochastic Behaviour.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-20
- A. Serrano Mena.** *Type Error Customization for Embedded Domain-Specific Languages.* Faculty of Science, UU. 2018-21
- S.M.J. de Putter.** *Verification of Concurrent Systems in a Model-Driven Engineering Workflow.* Faculty of Mathematics and Computer Science, TU/e. 2019-01
- S.M. Thaler.** *Automation for Information Security using Machine Learning.* Faculty of Mathematics and Computer Science, TU/e. 2019-02
- Ö. Babur.** *Model Analytics and Management.* Faculty of Mathematics and Computer Science, TU/e. 2019-03
- A. Afroozeh and A. Izmaylova.** *Practical General Top-down Parsers.* Faculty of Science, UvA. 2019-04
- S. Kisfaludi-Bak.** *ETH-Tight Algorithms for Geometric Network Problems.* Faculty of Mathematics and Computer Science, TU/e. 2019-05
- J. Moerman.** *Nominal Techniques and Black Box Testing for Automata Learning.* Faculty of Science, Mathematics and Computer Science, RU. 2019-06
- V. Bloemen.** *Strong Connectivity and Shortest Paths for Checking Models.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2019-07
- T.H.A. Castermans.** *Algorithms for Visualization in Digital Humanities.* Faculty of Mathematics and Computer Science, TU/e. 2019-08
- W.M. Sonke.** *Algorithms for River Network Analysis.* Faculty of Mathematics and Computer Science, TU/e. 2019-09
- J.J.G. Meijer.** *Efficient Learning and Analysis of System Behavior.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2019-10
- P.R. Griffioen.** *A Unit-Aware Matrix Language and its Application in Control and Auditing.* Faculty of Science, UvA. 2019-11
- A.A. Sawant.** *The impact of API evolution on API consumers and how this can be affected by API producers and language designers.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2019-12
- W.H.M. Oortwijn.** *Deductive Techniques for Model-Based Concurrency*

Verification. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2019-13

M.A. Cano Grijalba. *Session-Based Concurrency: Between Operational and Declarative Views*. Faculty of Science and Engineering, RUG. 2020-01

T.C. Nägele. *CoHLA: Rapid Co-simulation Construction*. Faculty of Science, Mathematics and Computer Science, RU. 2020-02

R.A. van Rozen. *Languages of Games and Play: Automating Game Design & Enabling Live Programming*. Faculty of Science, UvA. 2020-03

B. Changizi. *Constraint-Based Analysis of Business Process Models*. Faculty of Mathematics and Natural Sciences, UL. 2020-04

N. Naus. *Assisting End Users in Workflow Systems*. Faculty of Science, UU. 2020-05

