KATHOLIEKE UNIVERSITEIT
LEUVEN

Arenberg Doctoral School of Science, Engineering & Technology
Faculty of Engineering
Department of Computer Science

Operational Aspects of Type Systems

Inter-Derivable Semantics of Type Checking and Gradual Types for Object Ownership

Ilya SERGEY

Dissertation presented in partial
fulfilment of the requirements for
the degree of Doctor
in Engineering

November 2012

Operational Aspects of Type Systems

Inter-Derivable Semantics of Type Checking
and Gradual Types for Object Ownership

Ilya SERGEY

Jury:
Prof. Dr. Adhemar Bultheel, chair
Prof. Dr. Dave Clarke, supervisor
Prof. Dr. ir. Wouter Joosen, co-supervisor
Prof. Dr. ir. Frank Piessens
Prof. Dr. ir. Marc Denecker
Prof. Dr. Sophia Drossopoulou
(Imperial College London, United Kingdom)
Prof. Dr. Olivier Danvy
(Aarhus University, Denmark)

Dissertation presented in partial
fulfilment of the requirements for
the degree of Doctor
in Engineering

November 2012

D/2012/7515/118
ISBN 978-94-6018-584-7

To my parents, for their belief and boundless support in all my ups and downs.

To Lilia, for the three endless years of anticipation that are almost over.

Abstract

This dissertation presents a study of type systems as program semantics, their inter-derivation, and effects they have on a program execution.

Developing type systems for programming languages is a challenging task. When describing the formalism underlying a type system, the type system designer usually keeps in mind a series of denotational aspects specifying *what* the types mean. Indeed, the type system should be adequate and ensure a program property of interest. It should be also expressive: a well-developed formalism should reject as few good programs as possible.

Operational aspects of type systems and type checking formalisms that specify *how* types are inferred have received much less attention. This work makes two contributions with respect to the design of practical type systems and type-checking algorithms by taking an operational view on a static program semantics.

Multiple semantics for type checking and type inference are possible. However, before using one semantics instead of another, one needs to prove an appropriate correspondence theorem. The first contribution of this work is the elimination of this obligation. We demonstrate a constructive technique to inter-derive different operational semantics of type checking by applying a series of behaviour-preserving program transformation techniques. A pleasant consequence of employing program transformations is that no soundness and completeness theorems need to be proven for pairs of type-checking semantics: they are instead corollaries of the correctness of the inter-derivation and of the initial specification.

The second contribution of this work is a gradual version of a type system for strong encapsulation in object-oriented languages. Focusing on the impact of a rich type system on the dynamic operational semantics provides a mechanism for incremental migration of programs to use more expressive types. The described approach provides a tradeoff between annotation verbosity and dynamic checking of the encapsulation invariant. The technique is proven sound, evaluated on a well-studied code base and released as a publicly available compiler prototype.

Nederlandse Samenvatting

Operationele Aspecten van Typesystemen: Onderling Afleidbare Semantieken van Typecontrole en Graduele Types voor Object Eigendom

Dit proefschrift presenteert een studie van typesystemen als semantieken van programma's, hun onderlinge afleiding, en de effecten die ze hebben op de uitvoering van programma's.

De ontwikkeling van typesystemen voor programmeertalen is een uitdagende taak. Bij het beschrijven van het onderliggend formalisme van een typesysteem, houdt de ontwerper van het typesysteem meestal bepaalde denotationele aspecten in het achterhoofd, die de betekenis van de types bepalen. Het typesysteem moet effectief zijn en een bepaalde eigenschap afdwingen. Het moet ook expressief zijn: een goed ontwikkeld formalisme moet zo weinig mogelijk correcte programma's verwerpen.

Operationele aspecten van typesystemen en typecontrole formalismen die definiëren *hoe* types worden geïnfereerd hebben minder aandacht gekregen in de onderzoeksliteratuur. Dit werk levert twee wetenschappelijke bijdragen in verband met het ontwerp en de implementatie van praktische typesystemen en typecontrole algoritmes vanuit een operationele visie op een statische programmasemantiek.

Meerdere semantieken voor typecontrole en type-inferentie zijn mogelijk. Echter, alvorens een semantiek te kunnen gebruiken in plaats van een andere, moet men een correspondentieresultaat bewijzen. De eerste bijdrage van dit werk is de afschaffing van deze verplichting. We tonen een constructieve techniek om verschillende operationele semantieken van typecontrole van elkaar af te leiden met behulp van een reeks van semantiek-behoudende programmatransformaties. Een aangenaam gevolg van het gebruik van programma transformaties is dat er geen stellingen over correctheid en volledigheid moeten worden bewezen voor elke combinatie van twee

typecontrolesemantieken: deze volgen nu uit de juistheid van de onderlinge afleiding en van de oorspronkelijke specificatie.

De tweede bijdrage van dit werk is een graduele versie van een typesysteem voor sterke encapsulatie in objectgeoriënteerde talen. Door te focussen op de impact van een rijk typesysteem op de dynamische operationele semantiek, bereiken we een mechanisme om programma's incrementeel meer expressieve types te laten gebruiken. De beschreven aanpak biedt een afweging tussen een grote nood aan annotatie enerzijds en dynamische controle van de encapsulatie-invariant anderzijds. De techniek is correct bewezen, geëvalueerd op een goed bestudeerde codebase en uitgebracht in de vorm van een publiek beschikbaar compilerprototype.

Acknowledgements

One's journey towards a PhD degree is a series of fortunate meetings and the story of their consequences.

I would like to begin by thanking my advisor Dave Clarke. I vividly remember the moment we met at ECOOP in 2008 and the thrill I felt a few weeks later, when he invited me to join the DistriNet research group as a PhD student. Thank you, Dave, for the last four years, and for your inspiration and challenges.

Second, I would like to thank Olivier Danvy. I met Olivier at the Oregon Summer School on Programming Languages in 2009, where I was learning all the math in the 'verse. It was a turning point in my journey, which led to two immensely fruitful visits to Aarhus University in 2010 and 2011. Thank you, Olivier, for introducing me to the functional correspondence and to Calvin and Hobbes, and for your encouragement and advice.

I feel most fortunate for my dream-team PhD committee, and I sincerely thank each of its members, Wouter Joosen, Frank Piessens, Marc Denecker, Sophia Drossopoulou, Olivier Danvy and Adhemaar Bultheel both for serving in it and for our substantial scientific discussion at my preliminary defense, in October 2012.

From my co-authors, I learned a lot on communicating scientific ideas and shaping them up towards absolute clarity. I truly enjoyed multiple hours spent in discussions, writing and hacking with Dominique Devriese, Jan Midtgaard, Matthew Might and David Van Horn. It was a pleasure to collaborate with you, guys, and I am sure it will continue to be.

Submitting to conferences and journals elicited fantastic feedback from the anonymous reviewers, and I wish to express my gratitude to all of them, as well as to the program chairs and editors. I also wish to non-anonymously thank Roland Backhouse, Jeremy Gibbons and Shin-Cheng Mu for truly outstanding reviews.

Looking back, my appreciation of compilers, programming languages, and, of course, type systems grew out of attending the marvelous seminars organized by Dmitry Lomov for a fistful of nerdy students at Saint-Petersburg State University. I am grateful to my former boss Eugene Vigdorchik for mind-shaping discussions, which helped me to develop this appreciation to a true passion, even though he might not have fully realized this consequence back then. I am also thankful to Dmitry Boulytchev, Kira Vyatkina, Edward Hirsch, Alexander Nazarov, Anatoly Podkorytov and Andrey Terekhov for their great lectures and motivating discussions on the nature of scientific research.

I wish to thank Martin Odersky for hosting me at EPFL in summer 2009 and for introducing me to virtual classes. I am also grateful to the extraordinary researchers I had the pleasure to meet there: Tiark Rompf, Ingo Maier, Iulian Dragos, Philipp Haller, Lukas Rytz and Toni Cunei. In particular, I am thankful to Gilles Dubochet for discussions on object calculi and helping me to find a Swiss *couteau-économe* on very short notice.

I had a great time at Aarhus University, and my visit there would not have been half such a breathtaking experience without its insightful and fun people. Thank you, Ian Zerny, Jan Midtgaard, Peter A. Jonsson, Jakob G. Thomsen, Erik Ernst, Dominik Raub, Karl Klose, Christian Hofer and Tillmann Rendel, I am really missing the weekly Entropy meetings. Special thanks to Ann Eg Mølhave for making my visit smooth and unadventurous.

My three-months internship at Microsoft Research Cambridge and hacking into the guts of Haskell's demand analyser in summer 2012 were a true climax in my PhD journey. This internship put me in the privileged position to unleash my PL expertise to tackle a real-world problem. For this, I am sincerely grateful to my host Simon Peyton Jones. Thank you, Simon, for bringing me to the world of mainstream functional programming, where purely practical problems reside side-by-side with semantic beauty. I wish to extend my gratitude to Dimitrios Vytiniotis, who went far beyond the call of duty to a random student and offered me hours of exhaustive sessions in front of the white board, and to Simon Marlow, who was always tactful and friendly, answering piles of questions about GHC technicalities. I also wish to acknowledge researchers and guests of the Programming Principles and Tools group and, in particular, Andrew Kennedy, Mooly Sagiv, Noam Rinetzky, Gilles Barthe and Claudio Russo for giving a different perspective to the things I thought I knew. Of course, the internship entailed meeting a lot of brother-in-arms students: Dan Rosén, Sooraj Bhat, Abigail See, Nik Sultana, Anton Osokin and François Dupressoir, who generously shared the spirit of the inimitable atmosphere of research. In addition, I am grateful to Dr. Xiao Wang, who made me feel in Cambridge like at home.

Four years spent within the DistriNet research group brought me together with top-flight researchers, whom I am proud to call my colleagues and wish to thank for the rich intellectual environment they have created: Nelis Boucké, Mario Henrique Cruz Torres, Wouter De Borger, Adriaan Moors, Nick Nikiforakis, Pieter Philippaerts, Davy Preuveneers, Steven Op de beeck, Riccardo Scandariato, Raoul Strackx, Bart Vanbrabant and Stefan Walraven. It was my pleasure to be a part of the SecLang group and share my ideas on intense PLSIG meetings with Dominique Devriese, Willem De Groef, Marko van Dooren, Bart Jacobs, Adriaan Larmuseau, Jan Tobias Mühlberg, Marco Patrignani, Jan Smans, Dries Vanoverberghe and Frédéric Vogels, who were always happy to get involved into a deep technical argument. I wish to thank our incredible office managers Katrien Janssens and Ghita Saevels for carrying the burden of managing an office full of geeks, our always-ready-to-help administrators Esther Renson, Marleen Sommers, Denise Brams, Inge Vandenborne and Veronique Cortens for making my communication with the outer world easy and painless, and, of course, our Jack-of-all-trades system administration group experts Bart Swennen and Anita Ceulemans for our network infrastructure functioning without a hitch. In addition, I am thankful to my office mates Dimiter Milushev, Koosha Paridel, Frans Sanen, Rula Sayaf, Dimitri Van Landuyt, Arun Kishore Ramakrishnan and Ansar-Ul-Haque Yasar for making our office a friendly and joyful place to be.

Language classes at the Instituut voor Levende Talen were my secret sanctuary. For these wonderful hours of mere human communication I have to say *bedankt* to Benedicte Seynhaeve, Kirsten Fivez, Evelien Versyck, Mit Leuridan, Isabel Van Brussel, Florian Gerich, Alexander Lutz, Ania Maroszek, Marta Rubio Texeira, Norma Araceli Juarez Collazo, Alfredo Rial Duran, *¡gracias!* to Rosa De Trazegnies Otero, Bart Lauwereins, Anita Maes, Kevin Veulemans, Veronique Verreycken, Brigitte Wybo and *tak* to Dorien Smans.

My being in Leuven was comfortable and uneventful, no small thanks to José Proença, Kseniya Rogova, Olga Ivanchikova, Gloria Tiebout, Radu Muschevici, Stas Popkov, Natalia Sapun, Ilya Novikov and Maria Melnik.

This work would never have started without my prior experience in JetBrains with a team of world-beating professionals, my colleagues: Andrey Breslav, Konstantin Bulenkov, Nikolay Chashnikov, Alexander Chernikov, Irina Chernushina, Sergey Coox, Sergey Dmitriev, Eugenia Dubova, Michael Gerasimov, Peter Gromov, Dmitry Jemerov, Kirill Kalishev, Dmitry Krasilschikov, Vyacheslav Lukyanov, Olga Lukyanova, Anton Makeev, Alexey Makarkin, Egor Malyshev, Kirill Maximov, Sasha Maximova, Irina Megorskaya, Maxim Mossienko, Ann Oreshnikova, Vaclav Pech, Alexey Pegov, Eugene Petrenko, Mikhail Pilin, Alexander Podkhalyuzin, Ekaterina Polyakova, Andrew Serebryansky, Maxim Shafirov, Pavel Sher, Gregory Shrago, Sergey Vasiliev, Yegor Yarko, Sergey Zhukov and Eugene Zhuravlev. Thanks to each of you.

I am sincerely grateful to my friends, people I could always rely on, those who provided extensive support throughout my studies: Leonid Shalupov, Eugenia Iozefavischus, Slava Yamov, Dennis Ushakov, Anastasia Kazakova, Roman Chernyatchik, Oleg Shpynov, Sophia Nazarova, Oleg Tarasov, Elena Sivogolovko, Gleb Leonov, Karina Orlova and Anastasia Zhuravleva. Спасибо вам, друзья!

I should mention that my work was partially funded by the EU-funded FP7-project NESSoS, and I am thankful to people behind it, whoever they are, for the scholarship they provided and for the accompanying opportunity to make science.

Finally, I wish to thank Anindya Banerjee and Aleksandar Nanevski for offering me a post-doctoral position at IMDEA.

Contents

List of Figures

Introductory Words

In this dissertation we address the two issues in the design and implementation of type systems: the development of *correct* and *expressive* algorithms for type checking and the *gradual migration* of programs to use a richer type system.

Developing type systems for programming languages is challenging. When describing a formalism, for a type system a theoretician usually keeps in mind several aspects that a type system should possess. For instance, a type system should be *adequate* and ensure a program property of interest. There is also no doubt in the importance of the *expressiveness* of a type system: a well-developed formalism should reject as few good programs as possible.

Although adequacy and expressiveness are primary concerns of a type system designer, they are not the only characteristics that should be taken into the account. Thinking of type checkers in terms of implementation, i.e., *operationally*, one can wonder how efficient is one or another implementation of a given type-checking algorithm. Another valid concern of a practically-oriented programmer would be the possibility for *type-debugging* programs, i.e., given a type error, figure out what has actually has caused it and which particular constraints of the type formalism have not been satisfied. The expressiveness of a type system also has its cost: sophisticated program properties require verbose annotations in order to capture the programmer's intentions about the desired program behaviour and preserve a desired invariant in the whole program. This problem could be remedied by annotating parts of a program incrementally and relying on dynamic checks in not yet annotated parts, thus, performing a *gradual* migration.

Part I of this dissertation presents an operational view on type checking algorithms. Type systems are thought of not as traditional sets of derivation rules but as program interpreters, implementing a particular program semantics. This view makes it possible to implement different type-checking algorithms as program interpreters and make them the subject of program transformations. By applying a series of behaviour-preserving program transformation techniques, we show how to inter-derive a series

of operational formalisms of type checking. The pleasant consequence of employing program transformations is that no soundness and completeness theorems need to be proven for pairs of type-checking semantics: they are instead corollaries of the correctness of the inter-derivation and of the initial specification.

Part II of the dissertation is dedicated to taming the verbosity of type-systems for strong encapsulation in object-oriented languages. In order to overcome the rigidity and verbosity of an existing type system from the literature, we develop a corresponding *gradual* formalism, which provides a tradeoff between annotation verbosity and the safety guaranties of the original approach. In principle, only a small amount of annotations are required to indicate a programmer's intention with respect to the desired encapsulation property, allowing the compiler to instrument a program with necessary run-time checks. Given a fully-annotated program, the desired invariant for it can be checked statically at compile time. For partially-annotated programs, necessary dynamic checks will be emitted by a compiler, whenever a type information is insufficient for compile-time guarantee about the encapsulation invariant.

Both problems addressed share the operational view on type systems. In the first part of the dissertation type-checking formalisms are thought of as operational semantics, which makes them a subject of inter-derivation using program transformation techniques. In the second part, the type-based invariant checking is "projected" to the operational semantics of the underlying programming language, resulting in dynamic checks, when there is not enough information for type-level reasoning.

Part I

Inter-Derivable Semantics of Type Checking

Chapter 1

Introduction and Problem Statement

Μεταβάλλον ἀναπαύεται.
(In change is rest.)
HERACLITUS

Plus ça change, plus c'est la même chose.
JEAN-BAPTISTE ALPHONSE KARR

Type systems in programming languages are a well-established way to ensure fundamental properties of programs. The principle "well-typed programs do not go wrong" introduced by Milner [144] and evolved to "well-typed programs do not get stuck" by Wright and Felleisen [212] give an idea of some very basic types of these properties, such as program execution progress. However, modern type systems are also targeted to infer more specific program properties such as possible computational effects, non-interference, control-flow information and strictness [152, 191, 216, 98]. These enhancements inevitably affect the implementation of a type checking/inference algorithm, making it significantly harder to evaluate and to reason about. Therefore, when designing a type system as a form of program analysis, one should always distinguish between *the analysis* itself and *a program that implements the analysis*.

A well-designed type system makes a tradeoff between the expressiveness of its definition and the effectiveness of its implementation. The goal of this work is to

Expressions	e	$::=$	$n \mid x \mid \lambda x : \tau.e \mid e\ e$
Numbers	n	$::=$	*number*
Values	v	$::=$	$n \mid \lambda x : \tau.e$
Types	τ	$::=$	$\mathsf{num} \mid \tau \to \tau$
Typing environments	Γ	$::=$	$\emptyset \mid \Gamma, x : \tau$

Syntax

$$\text{(t-var)}\frac{(x:\tau)\in\Gamma}{\Gamma\vdash x:\tau} \qquad \text{(t-lam)}\frac{\Gamma,\ x:\tau_1\vdash e:\tau_2}{\Gamma\vdash \lambda x:\tau_1.e:\tau_1\to\tau_2}$$

$$\text{(t-app)}\frac{\Gamma\vdash e_0:\tau_1\to\tau_2 \quad \Gamma\vdash e_1:\tau_1}{\Gamma\vdash e_0\ e_1:\tau_2} \qquad \text{(t-num)}\frac{}{\Gamma\vdash \mathit{number}:\mathsf{num}}$$

Type checking rules

Figure 1.1: Syntax and Church-style type system of the simply typed lambda calculus.

bridge this gap and establish a systematic transition from an expressive definition to an effective implementation.

1.1 Type Checking as Program Semantics

Traditionally, type systems are described as collections of logical inference rules that are convenient to reason about. Such a representation is, however, not quite suitable to *observe* the process of type inference or type checking, which makes the debugging and optimization of typing algorithms complicated. In order to provide a more operational view to the procedure of checking and inference types, one can recall that type systems are often referred to as *static program semantics*. So, in fact, a type assignment procedure is just a way of thinking of what a program *should do* in terms of its types as opposed to expressions and values.

We start developing this simple observation by considering a type-checking procedure for the simply typed lambda calculus (STLC) as our running example. In STLC the only values are lambda-abstractions of the form $\lambda x : \tau.e$, variables or numeric literals, and expressions are either values or applications. Figure 1.1 describes the well-known typing rules for Church monomorphic static semantics of STLC [31]. Computationally, such a model, implemented straightforwardly in a functional

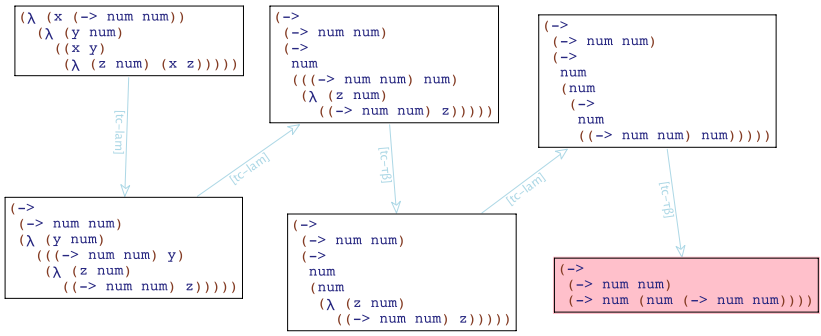

Figure 1.2: A sequence of type-checking reductions for an ill-typed term (1.1).

programming language, corresponds to a recursive descent over the inductively-defined language syntax, where a given expression is recursively traversed, so its type is derived when no typing errors occur. In the case of a typing error, however, it might be hard to track the true origin of the type error. For example, consider a small *ill*-typed program in STLC:

$$\lambda x : \mathsf{num} \to \mathsf{num}.\ \lambda y : \mathsf{num}.\ x\ y\ (\lambda z : \mathsf{num}.\ x\ z) \tag{1.1}$$

From the code, it might not be immediately clear why type checking is going to fail. Even more, the implementation of typing rules from Figure 1.1 as a recursive descent, without auxiliary instrumentation for debugging (e.g., logging), does not keep track of *well-typed* parts of the program that have been checked already. It is not clear, for instance, what steps have been taken before the type error occurred and which parts of the program have been typed successfully.

In order to remedy the problem of being able to reason about computational aspects, we need to pick more algorithmic representations of the type checking procedure. Below, we consider two alternative formalisms of type checking: a *reduction semantics* and an *abstract machine*.

1.1.1 Reduction semantics for stepping through the type checking

A reduction semantics for type checking was proposed initially by Kuan et al. [123]. Defined as a set of term-reduction rules, such a term-rewriting system gives an operational view on the static program semantics, which is convenient for debugging

complex type checking procedures, since the developer can trace each step of the type computation.

The reduction formalism treats expressions and types as elements of the same semantic space, so-called, *hybrid language*, where types can play the roles of values, although, plain (i.e., non-hybrid) values cannot be parts of types, which makes the system different from dependently-typed programming languages [132, 154, 161]. Informally, each step of execution in the evaluation strategy of type checking with reductions can be described as follows. First, the current program term is decomposed into a context computational context (i.e., "a term with a hole") and a redex (i.e., an elementary expression to be evaluated). Second, the contraction (i.e, evaluation) is performed (e.g., a lambda-abstraction is applied to its argument, which results in term substitution). Finally, the obtained result of contraction is "plugged" into the context's hole, and the newly recomposed term is returned. This sequence of steps is performed until a syntactically correct type is obtained. If this is not the case, the result of the evaluation with reductions is considered as "stuck" and characterizes a type error.

Figure 1.2 shows the sequence of steps for type checking preceding the type error while processing the ill-typed program (1.1).[1] The figure clearly indicates steps that have been performed during the type checking. For instance, in the first step, the lambda-expression $\lambda x : \mathsf{num} \to \mathsf{num}.(\ldots)$ is replaced by the hybrid expression $(\mathsf{num} \to \mathsf{num}) \to (\ldots)$, where $(\ldots)$ denotes a not-yet evaluated (i.e., not yet *type-checked*, in this terminology) hybrid part of the program. After five steps, the program reduces to a "stuck" hybrid term. By tracking the sequence of steps backwards, one can see that the cause of the typing error is an application of the non-function expression $(x\ y)$ of type num to the expression $\lambda z.\mathsf{num}.(\ldots)$. This is depicted by the innermost ill-formed type application `(num (-> num num))` in the last execution step in Figure 1.2.

Therefore, a reduction semantics for type checking enables one to track the process of type checking, making it possible to figure out the origin of a type error and its denotation in the form of an ill-formed type application. However, such a semantics still does not allow us to examine the "rest" of the type-checking process after the type error occurs. This aspect of the type checking is remedied using the following formalism.

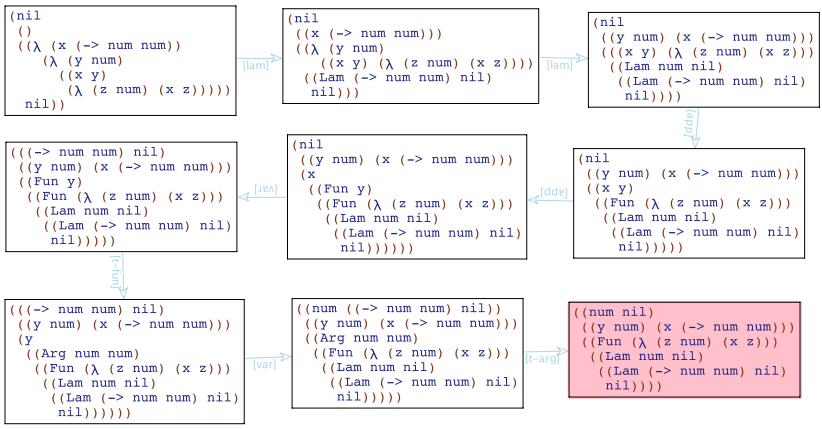

Figure 1.3: A sequence of abstract machine steps for an ill-typed term.

1.1.2 Abstract machine for context exploration and error recovery

When type-checking a program, one wants the type errors to be detected as early as possible. At the same time, the type checker should not fail after the first error: it should instead report it and recover to examine the rest of the program. As we have seen in the previous example, a reduction semantics for type checking, while it succeeds in depicting the sequence of steps during type checking, it is not particularly convenient for observing the remaining type computations after a type error has occurred.

In order to address this issue, we pick another operational formalism for type checking: a small-step SEC machine. The machine is a state transition system that deterministically maps one control triple to another. It is inspired by Landin's SECD formalism [124, 58], but lacks the last component of its control state—D, which we do not need, since there is no "dump" of the control flow in our machine. The control state of the machine consists of three components (registers): S—a stack for inferred types, E—a type environment binding variables with their types and C—a stack of control components, driving the machine. Given a control state, there are three possible scenarios how the machine can act:

1. The machine makes a step by mapping a state deterministically to another state, if a matching rule exists in the machine's semantics.

[1] We implemented the reduction strategy for type checking of STLC in PLT Redex framework [79], so the syntax of expressions and types is slightly different from the one in Figure 1.1. The full implementation can be found in Section A.1 of Appendix A.

2. If the state's component C is empty and S is not, the state is considered as a final state of execution and the type on top of the S component is taken as a result of type checking.

3. If there is no rule for the machine to make a step from a given state and the state is not a final one, then it is considered as a *stuck* state and corresponds to type-checking failure.

Without going any deeper into the formal description,[2] we invite the reader to take a look at the sequence of machine transitions for our example of an ill-typed program (1.1).[3] The last state of the sequence is indeed *stuck*, as the control stack component C is not empty and there is no rule to make any further transition; this corresponds to a type-checking error. By observing the state one can see that the typing environment at the moment the type error is occurred is $\{\langle y, \mathsf{num}\rangle, \langle x, \mathsf{num} \to \mathsf{num}\rangle\}$ and the error is caused by an attempt to process a non-function type num on top of the result stack S, where a function type is expected, which is indicated by the control element *Fun* $(\ldots)$ on top of the control stack. Just by observing the state, we have already learned a lot about the nature of the type error.

Furthermore, having a little bit more insight about how the present abstract machine works, the algorithm could *extract* the rest of the program to type-check from the control stack and continue the type checking! In this case, for instance, we could take the expression $(\lambda z : \mathsf{num}.x\ z)$ from the topmost control stack component and then restart the machine from the state

$$\langle \mathsf{nil}, \{\langle y, \mathsf{num}\rangle, \langle x, \mathsf{num} \to \mathsf{num}\rangle\}, (\lambda z : \mathsf{num}.x\ z, \mathsf{nil})\rangle.$$

This would give us the type of the expression.

From the example above, we have demonstrated that the abstract machine-based semantics for type checking provides freedom in reasoning not only about the sequence of steps taken during the typing algorithm, but also about the evaluation context and the environment at any moment of the execution, which provides a basis for implementation of error recovery in a straightforward way.

1.2 The Problem: Too Many Theorems to Prove

As we have shown in the previous section, different ways of representing type assignments are convenient for particular applications such as reasoning, effective

[2] Full semantic state-space and the transition function of the SEC machine are presented in Chapter 4.

[3] A PLT Redex implementation of the abstract register machine for type checking can be found in Section A.2 of Appendix A.

implementation or debugging the type checking procedure. However, before applying any of these semantics for the task of interest, some kind of correspondence between them should be proven in order to make sure that the operational formalism models original type checking rules in a sound and complete way.

Traditionally, appropriate soundness and completeness theorems need to be proven for this purpose. For example, the correctness of the correspondence between the natural type checking semantics (Figure 1.1) and the reduction semantics was proven by Kuan [121]. The result establishing a connection between the derivation-based static semantics and the type checking in the form of abstract machine is due to Hankin and Le Métayer [98].

In the meantime, one can think about other operational formalisms. For instance, the SEC machine, which we demonstrated in the previous section, is not the only possible abstract machine-like formalism. One may like to make use, for instance, of a variation of Felleisen and Friedman's CEK machine (Interpreter III of Reynolds [174]) for type checking [80]. And again, a formal correspondence theorem relating the natural static semantics and the abstract machine should be proven, similarly to the way it has been done for a family of traditional abstract *register* machines for the semantics of ISWIM [126] in the recent book by Felleisen et al. [79, Chapter 6].

What we would like to possess is a method to inter-derive different semantics of type checking in a way that the correspondence between them would be established *automatically* and correct *by construction*. Fortunately, such a method exists and this part of the present dissertation provides and investigates it in application to type checking algorithms.

1.3 The Method: the Functional Correspondence to the Rescue

When reasoning about formal semantics of some computation, one usually describes it by a set of rules or clauses employing some sort of a mathematically-structured *meta-language*. In fact, doing this, one assumes that the semantics of this meta-language is uniform and well understood. A natural consequence of this assumption is an attempt to employ an existing programming language with a well-understood semantics as such a meta-language. Informally, one can use computations to describe computations. Furthermore, if a semantics of a formalism of interest (e.g., type checking for STLC) is implemented in some actual programming language as a *program*, one can apply

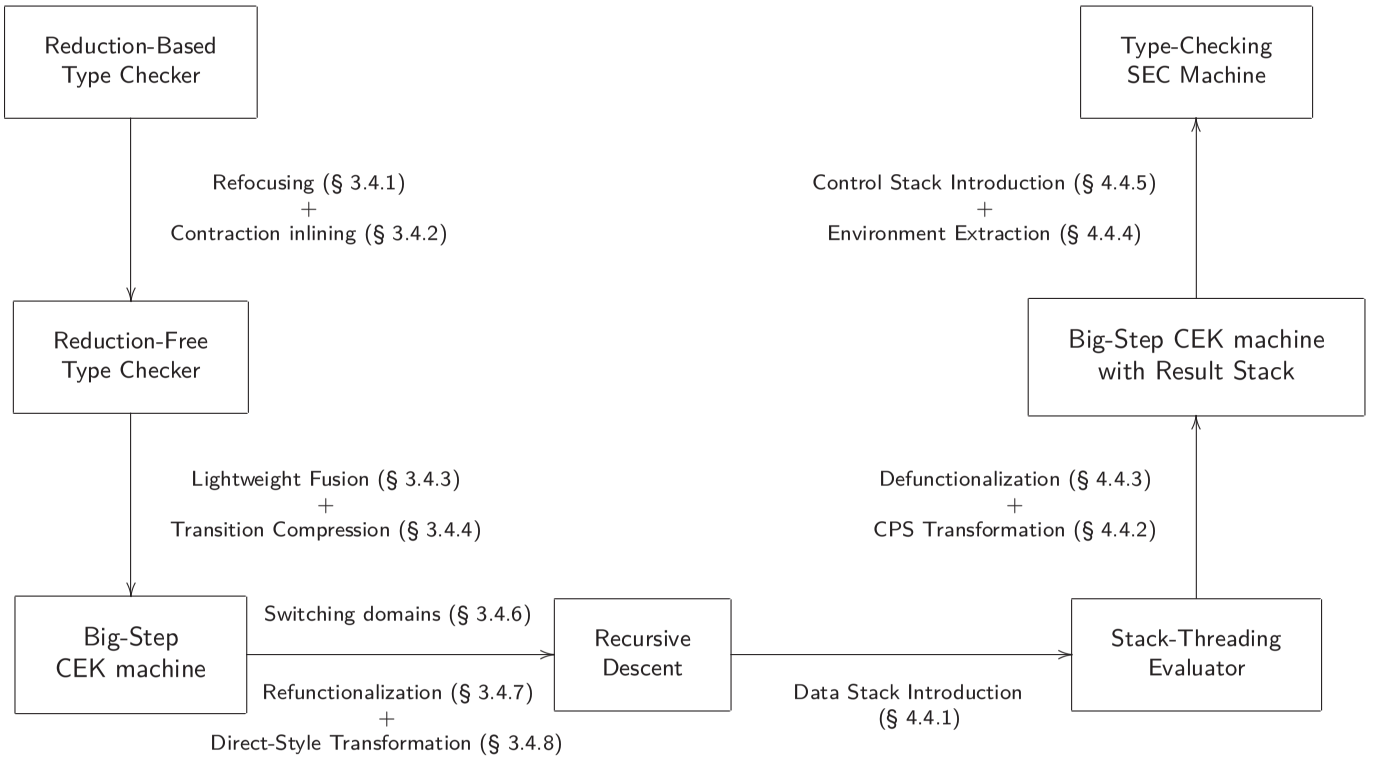

Figure 1.4: Inter-derivation of type-checking semantic artifacts.

a series of semantics-preserving transformations to this program.[4] The resulting program will correspond to *another* semantics of the investigated formalism, and what is left is only to identify this semantics and give it a name.

The method described above, which we are going to employ in order to *inter-derive* different semantics for type checking, is known as *functional correspondence* [51]. Following this approach, implementations of formal semantics in functional programming languages can be transformed into each other. In retrospect [2], the technique was initially pioneered by John C. Reynolds [174], who connected denotational semantics, natural semantics and big-step abstract machines using closure conversion, CPS transformation and defunctionalization as semantics-preserving program transformations in a meta-language. Later, Danvy et al. established the relation between small-step and big-step abstract machines using fusion by fixed-point promotion, and between reduction semantics and small-step abstract machines using refocusing and transition compression. Finally, the functional implementations of structural operational semantics and reduction semantics were related via CPS transformation and defunctionalization. We address the interested reader to Chapter 5 of the current dissertation or to Olivier Danvy's ICFP 2008 paper [51] for a brief state-of-the-art survey of applications of functional correspondence.

In the present work, we employ the toolset of functional program transformations investigated by Danvy et al. [2, 17, 52, 49, 62, 136] to establish the desired connection between various semantics of type-checking: traditional natural semantics in the form of recursive descent, reduction semantics by Kuan et al. [123] and an abstract machine by Hankin and Le Métayer [98]. The full derivation chain is depicted diagrammatically in Figure 1.4. Captions on arrows indicate the transformations employed and sections in chapters of the present work providing detailed explanation of the technique applied. In the course of the transformation we also derive a series of *novel* semantic artifacts of type checking, such as reduction-free type checker or a type checking big-step CEK-machine, as intermediate results of our derivations.

1.4 Main Contributions

The overall contribution of this part for the dissertation is a demonstation of the application of well-studied functional program transformations to inter-derive type-checking algorithms. Below, we explain its parts in detail.

[4] Here, *semantics-preserving* refers to the semantics of the chosen meta-language (as opposite to the semantics, implemented *in* the meta-language) and means that the behaviour of a transformed program in the meta-language is equivalent to the behaviour of the initial program.

A mechanical correspondence between type checking via reductions and via evaluation

Our first contribution is establishing a mechanical correspondence between two semantics of type checking: the reduction semantics by Kuan et al. [121, 122, 123] and the traditional formalization of type checking as inference rules, implemented in the form of a recursive descent [164]. We define the syntax of Kuan's hybrid language for type reductions and its semantics in terms of the chosen meta-language and then derive a traditional type-checking algorithm by applying semantics-preserving transformations, such as refocusing [52], lightweight fusion [158], refunctionalization [60] and others. The transformations we use are off-the-shelf and have already been proven to be semantics-preserving, so no correspondence theorems between the two semantics has to be proved. The method is illustrated in the setting of a simply-typed lambda calculus, but we also discuss its applicability for other formalisms, such as System F [93, 175] or Damas and Milner's algorithm $\mathcal{W}$ [144].

A mechanical correspondence between type checking via evaluation and via an abstract machine

Our second contribution is a natural continuation of the first and can be described as an establishing of a mechanical correspondence between type checking in the from of recursive descent [164] and type checking in the form of an abstract machine, reminiscent to Peter Landin's SECD formalism [124, 58]. The representation of a type checking algorithm in the form of an abstract machine has been known and was first observed by Hankin and Le Métayer [98] in the context of Jensen's strictness analysis for higher-order programs [113]. The derived correct *by construction* correspondence, we present, however, is novel. The method, from which the correspondence is established, is based on use of semantics-preserving program transformation techniques, mostly dual to ones we used for the first contribution: defunctionalization [62], direct-style transform [57] and others. To illustrate the approach, we use the simply-typed lambda calculus for the sake of simplicity. We also discuss possible extensions of the methodology to richer type systems.

A family of semantically equivalent artifacts for type checking

While providing a chain of derivations connecting the three semantics of type checking, we derive a series of *novel*, semantically-equivalent artifacts for type checking, as is shown in Figure 1.4. Among others, we extract a reduction-free type checker, a big-step type-checking CEK machine and a stack-threading type-checking evaluator. Therefore, systematically applying program transformations to well-studied

type checking semantics ends up giving a series of novel and interesting algorithms which, to the author's knowledge, have not been described elsewhere.

1.5 Notes on the Implementation

In order to implement different semantics for type checking, we have chosen an appropriate meta-language that is expressive enough to incorporate the toolset of a working semanticist seamlessly and at the same time provide a solid background to reason about the correctness of program transformations.

One possible candidate to this role would be the Racket programming language[5] with its extension PLT Redex, which we used to implement examples from Section 1.1. Defining semantics in PLT Redex is short, elegant and is very close to the mathematical notation as the reader can check by taking a look at Appendix A. However, the PLT Redex model is not quite suitable for our goals, since it

(a) is tailored for concise definitions of reduction-based semantics specifically, and

(b) relies heavily on the macro system of Racket, its host programming language.

This makes it complicated to perform program transformations. Also, the lack of a type system deprives us of the possibility to type-check the correctness of the encoded definitions, resulting in error-prone code, suffering from issues such as, for instance, non-exhaustive pattern-matching.[6]

We have chosen Standard ML (SML) [145] to the role of a metalanguage for the implementation and transformations. SML is a statically-typed, call-by-value language with computational effects.

For the sake of brevity we omit some of the program artifacts in Chapters 3 and 4, keeping only essential parts to demonstrate the corresponding program transformation. At each transformation stage the trailing index of all involved functions is incremented. The accompanying code for derivations in Chapters 3 and 4 is also available on GitHub:

`http://github.com/ilyasergey/typechecker-transformations`

[5] `http://racket-lang.org/`

[6] In fact, Racket provides an optional type system based on the notion of *occurrence typing* [202]. However, it requires expansion of all macros before type checking, which prevents a semanticist from assigning meaningful types to semantic artifacts when working with PLT Redex.

1.6 Outline

Part I of the dissertation describes the research the author conducted while visiting Aarhus University in 2010 and 2011 and is based on a combination of the following research papers and technical reports:

- Ilya Sergey and Dave Clarke. A correspondence between type checking via reduction and type checking via evaluation. *Information Processing Letters*, volume 112, issue 1-2, pages 13–20, January 2012. Elsevier.
- Ilya Sergey and Dave Clarke. A correspondence between type checking via reduction and type checking via evaluation. Accompanying code overview. *CW Reports, volume CW617*, 20 pages, Department of Computer Science, KU Leuven. January 2012. Leuven, Belgium.
- Ilya Sergey and Dave Clarke. From type checking by recursive descent to type checking with an abstract machine. In Claus Brabrand and Eric Van Wyk, editors, proceedings of the *Eleventh Workshop on Language Descriptions, Tools and Applications (LDTA 2011)*, pages 1–7. 26–27 March 2011. Saarbrücken, Germany. ACM.

The remainder of this part is structured as follows:

Chapter 2 – Background

The chapter provides necessary background about the basics of program semantics, funcional programming and functional program transformations.

In the first part of the chapter, we give a brief overview of the ways to describe computations and provide an informal survey of different semantic formalisms. We focus mainly on small-step operational semantics. We also enumerate a series of abstract machine-based formalisms along with their applications to the construction of sound program analyses using abstract interpretation.

In the second part, we recall the basic concepts of functional programming, such as the lambda calculus, first-class and higher-order functions and closures. We focus specifically on continuation-passing style and control operators for delimited and undelimited continuations as they are essential for the implementation of the derivations in further chapters.

In the third part, we introduce some well-studied functional program transformations. Readers familiar with the implementation of production-quality compilers [12,

162] will meet some old friends here: continuation-passing style and direct-style transformations, defunctionalization and refunctionalization, deforestation and many others.

Finally, we conclude the chapter with a toy example, employing most of the presented notions, tools and techniques: deriving a small-step, tail-call optimized stack-threading machine for computation of Fibonacci numbers from the standard recursive procedure, which corresponds to a big-step semantics.

Chapter 3 – From type checking via reduction to type checking via evaluation

The chapter presents a mechanical derivation of the correspondence between type checking via reductions and traditional compositional type checking implemented in the form of a recursive descent, filling the left part of the diagram in Figure 1.4. We describe an implementation of the corresponding semantic artefacts in the meta-language and the chain of the subsequent program transformations. The reader interested in non-essential implementation details is invited to take a look at the accompanying code in Appendix B or to the appropriate technical report [182]. This chapter corresponds to the first main contribution of this part of the dissertation and is a part of the article published in *Information Processing Letters*, volume 112, issue 1-2 [183].

Chapter 4 – From type checking via evaluation to type checking with an abstract machine

This chapter continues the story that began in Chapter 3 and presents a mechanical derivation of the correspondence between the compositional type checking algorithm in the form of a recursive descent and type checking in the form of an abstract machine, reminiscent to Landin's SECD formalism, thus, filling in the right part of the diagram in Figure 1.4. This chapter corresponds to the second main contribution of the Part I and is a part of the paper published in the proceedings of the Eleventh Workshop on Language Descriptions, Tools and Applications (LDTA '11) [181].

Chapter 5 – Related Work and Applications

In this chapter, we provide a survey of related work making use of Danvy's tool-chain for inter-deriving semantic artefacts. We focus mainly on recent research in semantics for concurrent and parallel programming. We also discuss possible applications of the

investigated type checking formalisms, for instance, for type debugging or type and effect analyses via abstract interpretation.

Chapter 6 – Conclusion and Future Work

The conclusion summarizes the findings of Part I. In this chapter, we elaborate on the obtained results and indicate directions for future research.

Chapter 2

Background

When describing the semantics of a program, we rely on mathematical notation as a metalanguage, the *interpretation* of which enables one to understand the meaning of the program. The next natural step is to define a semantics of a program directly by providing an *interpreter*, i.e., a program executing another program.

Indeed, any programming paradigm can be used to implement an interpreter. For instance, one can implement an interpreter in the form of a program in a functional programming language. Such a choice would have major consequences: it turns out that an interpreter written in the functional programming paradigm can be a subject of multiple behaviour-preserving transformations, and the result of each of these transformations will yield a *new* interpreter, delivering the same result, but having different computational properties. This observation has been taken as a basis of a technique, known as functional correspondence: program semantics are represented as interpreters and interpreters can be inter-derived using program transformations. Such a derivation corresponds to the constructive proof of equivalence of the corresponding semantics.

The goal of this chapter is to provide necessary background about the semantics of computations, functional programming and behaviour-preserving functional program transformations—three main components of the functional correspondence methodology. Readers familiar with these concepts are still encouraged to take a look on the text as it provides some important insights for the results described in Chapters 3 and 4.

2.1 Program Semantics and Abstract Machines

Program semantics is often defined as *the meaning of a grammatically correct program*. Such a broad definition gives one the freedom to construct different formalisms capturing different aspects of what *meaning* is. The semantics literature [153, 209] usually distinguishes three main classes of program semantics:

- *Denotational semantics* defines a meaning of a program by mapping it to a specific mathematical object in a particular domain (e.g., a function) [180].
- *Axiomatic semantics* defines the meaning of a program in terms of properties of the effect produced by executing the program [102].
- *Operational semantics* describes how to compute a program on some abstract machine and specifies what the result of a program is [166, 167].

Below, we give a brief overview of the first two approaches and focus on the last one.

2.1.1 Denotational semantics

Denotational semantics was originally introduced by Dana Scott and Christopher Strachey in order to reason about programs as state transformers [180, 153], answering the question, *what* a program is. The effect of program execution is, thus, modelled by relating a program to a mathematical function. Originally, denotational semantics was proposed to reason about imperative programs, for instance for computing the propagation of constants or reaching definitions. The semantics was applied later to establish fundamental results about functional programs as well. For instance, denotational semantics of the lambda calculus has been taken as a concrete domain to derive a series of type systems from the literature using abstract interpretation [39]. Although mathematically elegant, denotational semantics is, however, not particularly useful to deal with non-determinism and concurrency [164].

2.1.2 Axiomatic semantics

Axiomatic semantics defines the meaning of programs in terms of axioms and laws that particular constructs of the language should obey. Thus, the meaning of the program is a set of facts that can be *derived* from the program's shape according to the axioms. A canonical example of axiomatic semantics is Hoare triples [102]—the simplest axiom system to reason about effects of imperative programs in terms of changes in the state they perform. It is also typical of axiomatic semantics to

Expressions	e	$::=$	$x \mid \lambda x.e \mid e\ e$
Values	v	$::=$	$\lambda x.e$

Figure 2.1: Syntax of the untyped lambda calculus

ignore some aspects of program execution [153], which is not an obstacle for inferring *partial* correctness properties of programs. Referring to Pierce [164], we note that the "classic" axiomatic semantics experiences difficulties when reasoning about languages with procedures, which, however can be remedied by using more powerful formalisms, such as separation logic [176] and by annotating procedures explicitly [111].

2.1.3 Operational semantics

Operational semantics is of main interest for this work, as it focuses on computational aspects of the program (i.e, it answers the question *"How?"*) and usually can be directly mapped into an implementation of an interpreter. In the operational approach, the *behaviour* of a program is specified by a set of rules as a program would be executed on some sort of *abstract machine*. An abstract machine operates with *states*, which are typically tuples of one or several components. For instance, the state could be just an expression, and each machine *transition* maps it to another expression. More elaborate abstract machines are possible, and we enumerate some of them in Section 2.1.4 of this chapter.

As a demonstration of operational semantics we will use the untyped lambda calculus [32], a simpler version of the formalism employed in Chapter 1 (Figure 1.1). The syntax of the untyped lambda calculus is presented in Figure 2.1 and includes only three syntactic elements: variables, lambda-abstractions and applications. Values are represented by lambda abstractions only.

The meaning of a program in terms of operational semantics is its *result*, i.e., the final state (or a set of states) reached by a machine, when started with a program in its initial state. However, it is not always the case that an abstract machine reaches some sort of a final state. In this case, one might be more interested not in the existence of a final state, but in the *process* of computations, i.e., a sequence of steps that has been performed when computing the program. This fundamental difference in views on computations (result-oriented or process-oriented) led to two major subclasses of operational semantics.

$$\text{(B-Val)}\ \frac{}{v \Downarrow v} \qquad \text{(B-App)}\ \frac{e_1 \Downarrow (\lambda x.e) \quad e_2 \Downarrow v \quad \{v/x\}e \Downarrow v'}{(e_1\ e_2) \Downarrow v'}$$

Figure 2.2: A big-step operational semantics of the untyped lambda calculus

Big-step operational semantics

Big-step operational semantics, often also referred to as *natural semantics* or *evaluation dynamics*, defines a set of rules, that describe how to obtain the result of a program. What is important is that such semantics are usually recursive by nature and do not present the notion of *intermediate state* of a computation, but just describe how a particular program is mapped to its result, thus establishing a relation between an *initial* and a *final* state. When describing a big-step operational semantics, one traditionally follows the principle of semantic compositionality (Frege's principle): the meaning of a *whole* (program) is a function of the meanings of its (syntactic) parts together with the manner in which these parts are combined.

Figure 2.2 presents a version of a big-step semantics of the untyped lambda-calculus as the relation ($\Downarrow$). There are only two rules. The rule (B-Val) says that the value evaluates to itself. The rule (B-App) describes evaluation of ab application without specifying the computation order of a function and argument expressions (e_1 and e_2, respectively). Another good example of a big-step semantics is a set of rules for the type checking in a simply-typed lambda calculus as it is described in Figure 1.1. For instance, the rule (t-app) does not specify which of two application's subexpressions' types should be inferred first: the one of e_1 or e_2. Instead, the rule just relies on the results of type checking of e_1 and e_2, using the same type-checking relation ($\vdash$) "recursively".

Big-step operational semantics is often chosen as a way to formalize computations because of its simplicity and a declarative character, which results in a small number of inference rules which greatly simplifies the reasoning about programs. Reasoning is performed by induction on the shape of derivation trees following the standard induction principle:

1. Prove that a property of interest holds for all simple derivation trees by showing that it holds for the axioms of the transition system.

2. Prove that the property holds for all composite derivation trees: for each rule assume that property holds for its premises (the *induction hypothesis*) and then prove that it also holds for the conclusion of the rule.

The biggest disadvantage of the big-step approach is that it does not provide a representation of the computational sequence. As a result, using some rules one does not know what the next step in the execution will be, since the semantics just says how to evaluate the whole program at once. This characteristic is crucial for showing program properties, such as *execution progress*, which can be informally phrased as follows: if a program can be assigned a type (i.e., the program *type checks*), then it either has been already evaluated to a value (i.e., a final state), or can still "make a step". Obviously, the second part about progress (i.e., "making a step") cannot be formalized in a big-step settings, since there is no notion of step.[1] Moreover, the big-step semantics does not distinguish the failing or non-terminating programs and a program may fail to evaluate due to a missing case in the evaluation rules. The standard practice to remedy this issue is to provide a number of administrative error-handling rules and prove a *coverage* lemma, ensuring that all error cases have been handled [76]. The type soundness then would ensure that a well-typed program will not end up with an application of an error-handling rule.

Another domain that can hardly be addressed by a big-step formalism is reasoning about concurrent programs: having several processes, one cannot switch between them as there is no intermediate state for other processes to be suspended in.

To fix this issues one needs a kind of semantics that gives an explicit notion of a next evaluation step, so we come to the next section.

Small-step operational semantics

Small-step operational semantics, often also referred to as *structural operational semantics*, in its general view was formally introduced by Plotkin [166] in his seminal work *"A Structural Approach to Operational Semantics"* as a relation on program states.

Reduction semantics Speaking about program expressions as the simplest kind of possible program state, one usually defines a small-step formalism in terms of congruence and contraction rules. This approach is known in the literature as *reduction semantics*, such that each its step constructs a *new* expression.[2] For instance, for the untyped lambda calculus, a reduction semantics induced by the β-reduction relation ($\mapsto_\beta$) is described in Figure 2.3. $\{v/x\}e$ denotes a capture-avoiding substitution of a value v to the place of all occurrences of a variable x in the expression e. The rules that propagate the actual computation down the expression structure,

[1] However, a different formulation of a big-step semantics *co-inductively* allows to establish a weaker version of type soundness [128, 8], involving only the first part of the property about typing the final result.

[2] In the literature, the term "reductions" is traditionally used to denote various kinds of small-step semantics. In contrast, the term "evaluation" refers to big-step formalisms.

$$\text{(E-App1)}\ \frac{e_1 \mapsto_\beta e_1'}{(e_1\ e_2) \mapsto_\beta (e_1'\ e_2)} \qquad \text{(E-App2)}\ \frac{e_2 \mapsto_\beta e_2'}{(v\ e_2) \mapsto_\beta (v\ e_2')}$$

$$\text{(E-AppAbs)}\ \frac{}{(\lambda x.e)\ v \mapsto_\beta \{v/x\}e}$$

Figure 2.3: Congruence-based small-step operational semantics of the untyped lambda calculus [164].

such as (E-App1) and (E-App2), are referred to as *congruence rules*, and the rule (E-AppAbs) is a *contraction rule*.

One can notice that there is only one way to evaluate an expression in a seemingly non-deterministic case of an application: first, the function part e_1 is evaluated down to a value v and only then the argument part e_2 is evaluated. Once both sides are values, the contraction rule is applied. This order corresponds to the evaluation strategy known as *call-by-value* (CBV), meaning that the argument is evaluated down to a value before it is passed to a function.[3] This strategy describes precisely the order of computation, which is also implied that there is at most one "recursive call" to $\mapsto_\beta$ in rules in Figure 2.3.

Although the reduction semantics as we defined it is convenient for describing a simple syntax-driven transition system, it does not keep track of the expression decomposition process. Each step corresponds to a contraction rule (e.g., (E-AppAbs) from Figure 2.3), but after this, the expression should be *recomposed back* and the rules only state implicitly how to do it. What is more important is that the rules defined this way do not carry the *context* of computations around, which makes it complicated to describe concepts such as exceptions, advanced control operators (see Section 2.2.6) or context-oriented language features [34, 117].

In order to remedy this issue, Felleisen and Hieb suggested a description of congruence rules via *reduction contexts* [81]. A description of reduction contexts for the untyped lambda calculus is presented in Figure 2.4. Informally, a context is a "term with a hole" and its recursive definition points out explicitly which subexpression (a *redex*) should be contracted next. We denote syntactic decomposition of an expression e to a context E and a redex e' as $e = E[e']$ The semantic rules are drastically simplified and are described by only one rule (ContractAbs) that describes contraction within the context. The transition in such a semantics is represented by three procedures:

[3] Other strategies, such as *call-by-name* (CBN) and *call-by-need* are popular in implementation of lazy functional programming languages with memoization [163], however we do not focus on them in this work and refer an interested reader to the work [61].

$$\text{Contexts} \qquad E \quad ::= \quad [\,] \mid E\ e \mid v\ E$$

Reduction contexts

$$\text{(ContractAbs)}\ \frac{}{E[(\lambda x.e)\ v] \mapsto_\beta E[\{v/x\}e]}$$

Context-based small-step semantics rules

Figure 2.4: Reduction contexts and context-based reduction semantics for untyped lambda calculus.

decomposition of an expression into a context and a redex, contraction, and a recomposition of an expression from the result of the contraction (*contractum*) and a context. The procedure is repeated and a possible valid result of the program is a value. This reformulation of a standard reduction semantics is often called an *abstract syntax machine* [79] as it is, indeed, the simplest abstract machine with a one-component state—an expression e and a fairly "big" execution step: *decompose – contract – recompose*.

A note on terminology There is a small controversy in the terminology in a semantics literature: sometimes a term *structural operational semantics* refers to the semantics where each step is either an elementary decomposition or recomposition (i.e., "remove" or "put" one level of a reduction context), or a contraction. In contrast, the *reduction semantics* considers the whole chain decompose – contract – recompose as one step [51].

Despite the simplicity of the reduction semantics, it is not the only possible for programs. For instance, one can imagine a possibility of implementing another evaluation strategy, when a body of a lambda-abstraction $\lambda x.e$ is not traversed when applied due to the substitution $\{v/x\}$. Instead, one would like to *delay* the use of the value v before the actual moment the variable x is examined in the expression e. In order to do so, when decomposing an expression to a context and a redex, one would need to maintain an extra component, namely, a local *environment*, as a mapping from variables to values. In contrast with an abstract syntax machine, such a semantics would correspond to a much more fine-grained abstract machine, whose step is either a contraction or "peeling away" of one layer of a context, and a state has now two components: a context and an environment. This is the subject of the next section.

2.1.4 Abstract register machines

Literature on program semantics often emphasizes a category of *abstract register machines* defined as of small-step operational semantics, whose state contains several components (i.e., registers). By abusing terminology, we will further refer to abstract register machines simply as to *abstract machines*.[4] Most of the notable abstract machines are named according to the abbreviated names of their state components. In this section, we will outline some essential exemplars of abstract machines and indicate their applications. A broader overview of abstract register machine formalisms can be found in Felleisen et al.'s book [79, Chapter 6].

Landin's SECD machine

Landin introduced the SECD machine as the first implementation of the lambda calculus as a programming language [124]. Following from its name, the state of the SECD machine consists of four components. The SECD machine is stack-based, so the arguments and results are stored in a result stack component S. The current variable environment component that maps variable names to their values is second and is denoted by E. The control component C contains a stack of control elements and its head always points to the next element to be processed. Finally, the last component, D for *dump*, is used to store the stack of temporary components from other registers and can be used, in particular, to implement jumps in a program.

Figure 2.5 provides a definition of the state-space elements and the transition relation $(\Rightarrow_{\text{SECD}})$ of the simplest SECD machine for the untyped lambda-calculus. The notation $\overline{x \mapsto clo}$ stands for a possibly empty non-ordered set of mappings from variables to closures, and $\rho[x \mapsto (v, \rho')]$ denotes an *update* of and an environment ρ for a variable x. The initial state for a SECD machine for an expression e and an environment ρ is a quadruple $\langle \mathsf{nil}, \rho, \mathit{Term}(e) :: \mathsf{nil}, \mathsf{nil} \rangle$. The final configuration corresponds to a state $\langle clo :: \mathsf{nil}, \rho, \mathsf{nil}, nil \rangle$ for some closure clo and value environment ρ.

A reader can notice a similarity between the description of the SECD machine and the abstract machine demonstrated in Chapter 1. A stack-based machine of this shape will play a key role in the derivation presented in this work and will be considered in detail in Chapter 4.

[4] Sometimes we will use the term "big-step abstract machines" when referring to big-step operational semantics with a multi-component initial state.

Expressions	e	$::=$	$x \mid \lambda x.e \mid e\ e$
Value environments	ρ	$::=$	$\overline{x \mapsto clo}$
Closures	clo	$::=$	(ρ, x, e)
Control elements	c	$::=$	$Term(e) \mid Apply$
Results stacks	S	$::=$	$\mathsf{nil} \mid clo :: S$
Control stacks	C	$::=$	$\mathsf{nil} \mid c :: C$
Dump stacks	D	$::=$	$\mathsf{nil} \mid (S, \rho, C) :: D$

SECD machine state-space

$$
\begin{array}{rcl}
\langle clo :: \mathsf{nil}, \rho', \mathsf{nil}, (S, \rho, C) :: D \rangle & \Rightarrow_{\text{SECD}} & \langle clo :: S, \rho, C, D \rangle \\
\langle S, \rho, Term(x) :: C, D \rangle & \Rightarrow_{\text{SECD}} & \langle clo :: S, \rho, C, D \rangle \text{ where } clo = \rho(x) \\
\langle S, \rho, Term(\lambda x.e) :: C, D \rangle & \Rightarrow_{\text{SECD}} & \langle (\rho, x, e) :: S, \rho, C, D \rangle \\
\langle S, \rho, Term(e_1\ e_0) :: C, D \rangle & \Rightarrow_{\text{SECD}} & \langle S, \rho, Term(e_1) :: Term(e_0) :: Apply :: C, D \rangle \\
\langle (\rho', x, e) :: clo' :: S, \rho, Apply :: C, D \rangle & \Rightarrow_{\text{SECD}} & \langle \mathsf{nil}, \rho'[x \mapsto clo'], Term(t) :: \mathsf{nil}, (S, \rho, C) :: D \rangle
\end{array}
$$

Transition rules

Figure 2.5: Landin's SECD machine for the untyped lambda calculus

$$
\begin{array}{lrcl}
\text{Values} & v & ::= & \lambda x.e \\
\text{Environments} & \rho & ::= & \overline{x \mapsto \langle v, \rho \rangle} \\
\text{Continuations} & \kappa & ::= & \mathbf{mt} \mid \mathbf{ar}(e, \rho, \kappa) \mid \mathbf{fn}(v, \rho, \kappa)
\end{array}
$$

Variable environments and continuations

$$
\begin{array}{rcl}
\langle x, \rho, \kappa \rangle & \Rightarrow_{\text{CEK}} & \langle v, \rho', \kappa \rangle \text{ where } \rho(x) = (v, \rho') \\
\langle (e_1\ e_2), \rho, \kappa \rangle & \Rightarrow_{\text{CEK}} & \langle e_1, \rho, \mathbf{ar}(e_2, \rho, \kappa) \rangle \\
\langle v, \rho, \mathbf{ar}(e, \rho', \kappa) \rangle & \Rightarrow_{\text{CEK}} & \langle e, \rho', \mathbf{fn}(v, \rho, \kappa) \rangle \\
\langle v, \rho, \mathbf{fn}(\lambda x.e, \rho', \kappa) \rangle & \Rightarrow_{\text{CEK}} & \langle e, \rho'[x \mapsto (v, \rho)], \kappa \rangle
\end{array}
$$

The CEK machine transition relation

Figure 2.6: The CEK machine for the untyped lambda calculus

CEK machine

CEK machine was introduced by Felleisen and Friedman [80] as a realistic model of the context-based reduction semantics with explicit variable environments that is amenable to efficient implementation.

Instead of substitution, which is typical for β-reduction, the CEK machine uses environments and closures (i.e, values paired with variable environments) to model substitution. The state, thus, consists of three components: a control string (i.e., an expression) C, the current environment E and a continuation K, a list-like inductive data type representing the "rest of computation" (see Section 2.2.5). Figure 2.6 provides a definition of environments, continuations and the transition relation ($\Rightarrow_{\text{CEK}}$) of the CEK machine. The initial state for an expression e with no free variables is a triple $\langle e, \emptyset, \mathbf{mt} \rangle$. A valid final state of the CEK machine is of the shape $\langle v, \emptyset, \mathbf{mt} \rangle$.

One can notice strong similarities between the context-based abstract syntax machine and the CEK machine. For instance, reduction contexts E (Figure 2.4) can be represented by continuations inside-out as follows: $[\,]$ is represented by **mt**; $E[([\,]\ e)]$ is represented by $\mathbf{ar}(e', \rho, \kappa)$ where ρ binds free variables of e' to represent e and κ represents E; finally, $E[(v\ [\,])]$ is represented by $\mathbf{fn}(v', \rho, \kappa)$, where ρ binds free variables of v' to represent v, and κ represents E. A mechanical proof of this correspondence is the subject of numerous works [2, 3, 4] and the same proof widely used in Chapter 3 of the present dissertation.

Small-step abstract machines and control-flow analysis

Effective implementation is not the only reason of exploring different abstract machine-based formalisms. Recently, abstract machines have acquired significant attention as a simple way to construct numerous static analyses via the method known as "*abstracting abstract machines*" [204, 139]. The reason of this is twofold. First, abstract machines are instances of small-step operational semantics, and, therefore, provide explicit notions of an intermediate state and a step of computation. Second, abstract register machines possess one more important characteristic: by the structure of the state, they make *explicit* the representation of important computational aspects, such as *bound variables* (thanks to the environment component E of SECD or CEK machines) or *continuations* (by employing the K component of the CEK machine). In order to get exhaustive information about bound variables and control flow in a higher-order program, one just needs to collect *all* reachable states during the program execution. This sort of computation is described by a notion of *collecting semantics* [43].

Unfortunately, the problem of collecting all reachable states is undecidable in general as it is equivalent to solving the *halting* problem. The usual approach in the literature is to compute an *approximation* of the set of all reachable states by the method known as *abstract interpretation* [40, 42]. In application to abstract machines, this would mean defining a *sound* abstraction for concrete states by some sort of *abstract states* in the way that each abstract state may correspond to multiple concrete states. By designing the space of abstract states in the way such as the space is finite and proving the sound approximation property (known as a *Galois connection* [137]), one can employ an *abstract collecting semantics* as a collecting semantics built on top of the "abstract" abstract machine (i.e., one operating with abstract states). Thanks to the finiteness of the abstract state-space, the abstract collecting semantics is always computable.

Employing this methodology implies a small refactoring of a state-space of a "concrete" abstract machine in order to avoid recursive definitions (e.g., environments and continuations, as in Figure 2.6). Van Horn and Might describe the systematic methodology for doing this [204, 139] by resolving the circular dependencies in a concrete state-space. Sergey et al. show how to employ monadic comprehensions to implement both "concrete" and "abstract" versions of an abstract machine in the same framework [184]. The "abstracting abstract machines approach" has also been recently extended for the object-oriented paradigm [141] and is used as a unifying technique to compare families of points-to analyses for Java-like languages [192].

2.1.5 Computational look on operational semantics

The observant reader could have already noticed that there are significant similarities between all operational formalisms and the way a programmer usually implements an interpreter. This section emphasizes and summarizes these similarities, which are a crux of our study.

From the intuition behind a big-step semantics, a functionally-oriented reader can easily recognize the form of *catamorphism*: big-step operational semantics is simply a *fold* over a program's syntactic tree thought of as a data type. A fold is usually implemented as a recursive descent, an *evaluator* that calls itself on subparts of a traversable structure, i.e. defined by structural induction on source terms.

An abstract machine is also an interpreter. However, in contrast with *evaluators*, abstract machines are implemented by providing a *transition function* and a *driver loop* function. A transition function defines a step logic of the semantics. A driver loop function performs the step iteration and specifies what a final state is.

In these terms, the states of an abstract machines are simply *inputs* and *outputs* of the transition function. As has been pointed out, *any* small-step semantics corresponds to a sort of abstract machine.

Therefore, we have an informally defined correspondence between operational formalisms and their computational counterparts:

- big-step operational semantics $\approx$ evaluator $\approx$ *recursive descent*;
- small-step semantics $\approx$ abstract machine $\approx$ *transition* and *driver loop* functions.

We believe that this insight is sufficient for the reader to recognize both types of formalisms from the shape of their implementations.

2.2 Elements of Functional Programming

Traditionally, one thinks of functional programming as a *declarative* programming paradigm. In the world of contemporary mainstream programming, functional programming is represented by various dialects of programming languages such as Lisp, ML and Haskell. In contrast with the *imperative* paradigm, represented by such languages as Pascal, Java and C, where a *program* is thought as a sequence of steps to compute the desired results (i.e., informally, the program is an answer to the question "How?"), in a declarative language the programmer describes a desired result itself

(i.e., the program is an answer to the question "What?"), assuming the solution of the problem to be inferred from the problem description.

In functional programming, the declarative aspect is reached by thinking of a program as of a composition of expressions. Programs written in such a way, allow one to understand them and reason about them *compositionally*. This approach to program development has proven to be useful in multiple application domains, such as compiler construction, financial computations, constructing electric circuits, scientific computing and many others. The literature on applications of functional programming counts thousands of remarkable works, and we recommend the interested reader to start from the seminal paper of John Hughes [106]. In the present work, we are going to use functional programming as a way to define interpreters and reason about them.

This section provides a short overview of the main concepts of functional programming that we will use as the tool for describing the semantics of computations and, eventually, for establishing the results in Chapters 3 and 4.

2.2.1 The lambda calculus

The lambda calculus is the core underlying formalism of all functional programming languages. It was originally inspired by the works of Moses Schönfinkel and was formally introduced by Alonzo Church [32] as a formal system to study compositions of computations. The simplest form of the lambda calculus contains only three syntactic constructs, also referred to as *terms* or *expressions*: variables, lambda-abstractions (as a representation for functions), and applications. Initially, the lambda calculus was introduced in its untyped form. Later, the simply-typed lambda calculus was introduced as a formalism ensuring extra properties of terms [31]. The polymorphic lambda calculus, also known as System F, one of the most widely-used typed formalisms on top of the lambda-calculus, has been later independently discovered by Girard and Reynolds [93, 175].

The idea of basing a programming language on the lambda calculus was pioneered by Landin [124]. He also pointed out the resemblance between contemporary programming language concepts and Church's lambda calculus [125], stating that lambda abstractions correspond to procedure declarations and applications correspond to procedure calls. The lambda calculus has since been taken as the basis of several modern programming languages, such as Standard ML [145], Scheme [196], Haskell [163], Objective Caml [129], F# [198] and Scala [156].

2.2.2 Key concepts of functional programming

There is no strict notion of what a functional programming language is. However, most of the languages holding the lambda calculus as their core and considered as functional ones, share the following traits:

- *Programs are expressions.* Since a program is constructed as the composition of expressions, each expression yields a value after being evaluated, which, in its turn, might be consumed by other expressions. It is *expressions* that are considered as building blocks in contrast with *statements* in imperative programming languages, which are treated according to their side effects.
- *Functions are first-class citizens.* Functions are full-fledged values in the language: they can be passed along as arguments to other functions. The latter are normally referred to as *higher-order functions*. Sometimes, higher-order functions not depending on the environment, are referred to as *combinators* and form the basis of function-oriented techniques such as combinator parsing [107].[5]

2.2.3 Closures

Since functions can be created and passed as values in functional languages, the function pointer itself is not sufficient to represent a function, as the function could have been created in a specific environment. In fact, values bound to the function's free variables (i.e., those, not bound by the function's parameters) may outlive their point of declaration, for instance, when the function is returned as a result of another function. As an example, consider the following simple code fragment in Standard ML:

```
fun mult n
  = let fun g x = x * n
    in g end
```

The result of the call `mult 42` will be the function that multiplies its argument by `42`. In this case, the free variable `n` in the body of the function `g` will be bound to `42`.

In order to consider the functions along with the *environment* they are created in, Landin introduced the notion of *closure* to correctly represent functional values [124]. A closure is a term paired with an environment, representing the values bound to

[5] The notion of *combinator* was independently introduced by Moses Schönfinkel and Haskell Curry, and plays a key role the correspondence between lambda calculus and *combinatory logic* [45]. It is also essential for effective implementation of lazy functional programming languages [162, Chapter 13].

free variables in the term's body.[6] The resulting pair component is *closed* in the sense that all free variables in the term component now refer to bound variables in the environment component.

Closure-based semantics of programming languages is a well-known alternative to a traditional substitution-based semantics [164, Chapter 7]. The key difference between them is that the former performs a lookup into the environment component of the closure, once a value of a free variable is evaluated, whereas the latter avoids dealing with free variables, eagerly performing a capture-avoiding substitution of values according to the reduction strategy. This difference is often used as motivation to introduce closures as a way to implement computations more efficiently [79], as substitution of values is then delayed to the moment of actual examination of the corresponding variables.

Most contemporary mainstream programming languages have some analogue of closures. For instance, the Java programming language [95] supports both named and anonymous inner classes that can be thought as closures in the sense that objects (values) generated from these definitions (similarly to lambda-abstractions) can refer to variables in their enclosing scope. In order to compile these inner definitions, a technique, similar to *lambda-lifting* in functional languages (see Section 2.3.8) is applied [65], whereby captured references to the outer class' environment are converted to object fields analogously with a closure's environment component.[7]

2.2.4 Tail calls and tail-call optimization

One well-known programming technique in functional style assumes implementation of most functions in the *tail-call* style. The function call is in the *tail-call position* of its caller function if it is the *last* expression of the caller to be evaluated before the caller returns the result. If the function performs a tail-call of itself, it is referred to as tail-recursive. Tail calls are essential for a program compilation technique known as *tail call optimization* [196], which consists of the removal of the caller's frame from the call stack once the callee in the tail position has been invoked. This allows, in particular, the implementation of a tail-recursive function as a loop thereby avoiding program stack overflow errors [38].

From the semantical point of view, the evaluation of the program, whose functions are all in tail-call positions forms a *sequence*, in contrast with an *evaluation tree* in the non tail-call case, where all constituents of the expression should be evaluated before the expression can be computed. The use of tail calls is a great way to *sequentialize*

[6] In fact, *values* now are defined as closures, which gives a circularity in the definition and poses some known challenges when interpreting the semantics [204, 139].

[7] At the present moment, this compilation technique is being discussed as a strategy to implement first-class functions in the upcoming Java 1.8 (JSR 335).

computations in a functional program, which brings us to the following essential concept from functional programming.

2.2.5 Continuation-passing style

The existence of first-class functions as an inherent component of functional programming makes it possible to parametrize a function with *the rest of its computation*. This programming technique is known as *continuation-passing style* (CPS) and is usually implemented by making functions accept an additional parameter for a *continuation*. In CPS, when the intermediate computation step is complete, the continuation parameter is applied to its result (i.e., the result is passed to the rest of computation as an argument). Tail calls in functions pass the continuation parameter unmodified, whereas non-tail calls augment the continuation to name the result of the call and continue the computation with the named intermediate result. The listing below presents two implementations of the factorial functions implemented in Standard ML: the traditional one in direct style and the one in CPS.

```
(*  Implementation of the factorial in the direct style  *)
fun fact n
  = if n = 0 then 1 else n * fact (n - 1)

(*  Implementation of the factorial in CPS  *)
fun fact_cps (n, k)
  = if n = 0 then k 1
    else fact_cps (n - 1, fn x => k (x * n))

(*  Running the CPS factorial by a non-CPS function  *)
fun run_fact n = fact_cps (n, fn x => x)
```

Note that in order to run the program in CPS, one should provide an *initial continuation* that will be used as a basis to process the "outermost" result of the called function. In the example above, this is performed by a function `run_fact` that pass the identity function `fn x => x` for `fact_cps` as an initial continuation.

Despite its simplicity, CPS is used by semantics engineers and compiler engineers. The former gain from the fact that the program is CPS is linearized: the program itself encodes the evaluation orders with all intermediate results named, which allows on to represent computations in CPS as a *sequence* in contrast with the *evaluation tree* in the general case thereby producing an easy semantics as the basis for the deriving of semantics-based artifacts, such as systematically constructed control-flow analyses [137, 138, 139, 179, 184, 186]. Moreover, the initial formulation of the denotational semantics of imperative languages as state transformers was done in the form of CPS in order to manage non-local control operators, such as function returns and the raising of exceptions [153].

For practical purposes, CPS is used as an intermediate form for further program transformations and optimizations [12, 89]. It is easy to notice that in the *pure* CPS form, each call is a tail call, so each function invokes another function or its continuation parameter as the only expression in its body. Such form proved, however, impractical as it would require rewriting *all* program constructs to CPS [55]. One typically keeps pure and total primitives (i.e., defined on all arguments) in the original direct style. For instance, the mathematical primitives such as $>$, $-$, $+$ and others are left unmodified as well as the conditional expression, which otherwise would accept two continuations, corresponding to each branch.

CPS, Administrative Normal Form and Static Single Assignment

As it has been noted, CPS allows the representation of computations of a functional program sequentially, which makes it useful as a clean and simple formalism. However, other ways of sequentialization of computations are possible. For instance, Sabry and Felleisen [178] suggested an untyped version of the *monadic normal form* by Moggi [100, 148], in which all intermediate results are assigned to variable and then the variables are used in the subsequent computations. This form is usually referred to as *administrative normal form* (ANF), following Flanagan et al. [87].

Since in CPS each intermediate result is assigned to the continuation parameter, one can notice the similarity between CPS and the form known as *static single assignment* [149], in which each variable in the program is assigned only once. The correspondence between CPS and SSA has been formally shown by Kelsey [115]. He has also shown that some CPS programs cannot be compiled to SSA; however these are not obtained by the regular CPS transformation (see Section 2.3.1).

2.2.6 Control operators in higher-order languages

Writing programs in continuation-passing style gives the programmer the explicit possibility to manage the control flow of the program, since continuations are now represented as functions, which are *first-class*. For instance, one can compose a continuation with another function (as it is done in the factorial example above) or simply drop it, which would correspond to the explicit interruption of the control flow, which is normally implemented in imperative languages using exceptions, explicit returns and control operators such as `break` and `continue`.

However, as has been discussed previously, it is not always convenient to write the whole program in CPS, as this style might introduce a fair amount of administrative code for proper continuation management. In order to overcome this issue, multiple *control operators* have been introduced to higher-order languages, making it possible

to manage the first-class continuations. Nowadays, control operators are implemented in the majority of modern higher-order functional languages, such as Scheme and Standard ML of New Jersey. There are two main ways to classify first-class continuations. First, continuations can be either *undelimited* (i.e., they represent the entire rest of the computation), or *delimited* (i.e., the represent only a fragment of the rest of computation). Second, continuations can be either *jumpy* or *pushy*. *Jumpy* continuations are similar to jumps in imperative languages, as they are resumed by aborting the current continuation. *Pushy* continuations are similar to non-tail calls implemented in CPS, as the are resumed by composing them with the current continuations Traditionally, in practice only two combinations of first-class continuations are considered: undelimited and jumpy, and delimited and pushy.

Undelimited continuations

First-class undelimited continuations are made available in functional programming languages via control operators such as `call/cc`. Applications of first-class undelimited continuations include, among others, backtracking, modelling coroutines, lightweight processes and non-local exits from recursion [82]. Historically, `call/cc` is a successor of Landin's J-operator [127], which was introduced in order to connect Algol 60's goto statements [125] with the semantics of lambda calculus [124].

We show the intuition behind the control operators for undelimited continuations with the following example involving Scheme's `call/cc` primitive. Consider the following arithmetic expression:

$$1+2+\boxed{3\times 4}$$

one implicitly assumes that the first subexpression to be evaluated is the emphasized part 3×4. The rest of the expression is the *continuation* of the first expression's evaluation. It can be informally represented as a term with a "hole": $1+2+\square$. Scheme's `call/cc` operator gives a possibility to *capture* this continuation and use it as a function, whose argument is substituted for the placeholder "hole" $\square$. For instance, consider the following pseudo-Scheme code:[8]

$$1+2+\boxed{\texttt{call/cc}\ (\lambda k.\ 8\times(k\ 5))} \tag{2.1}$$

The captured continuation, denoted by the light grey box, is bound to the variable k, i.e., $k = \boxed{1+2+\square}$. The call of `k` within the `call/cc` construct *aborts* the

[8] The example can be implemented in Racked as
`(+ 1 2 (call/cc (lambda (k) (* 8 (k 5)))))`,
given the package `racket/control` is imported.

evaluation of the outer continuation $1+2+\square$ and instead uses it as a function k in the body of the lambda-expression passed to `call/cc`. Traditionally, `call/cc`'s given continuation is jumpy, that is, the rest of the current continuation, $8 \times \square$, is dropped once k is applied. Thus, the expression (2.1) is turned into the expression $k\ 5$, which evaluates to

$$1+2+\ 5$$

and yields 8 as a result.

Delimited continuations

Unlike operators similar to `call/cc`, control primitives for *delimited continuation* management provide a mechanism to capture only the part of the rest of computation, use it as a first-class continuation and *compose* the result with the current continuation. Delimited continuations were introduced independently by Felleisen [78] and by Danvy and Filinski [53]. The way the composition with the current continuation is implemented determines whether the delimited control operators are *static*, i.e., compatible with CPS (e.g., Danvy and Filinski's `shift` and `reset`) or *dynamic* (e.g., Felleisen's `control` and `prompt`).

We demonstrate the use of Danvy and Filinski's control operators `shift` and `reset` by the following simple example with arithmetic expressions:[9]

$$1+\ \texttt{reset}\ (2+\ \texttt{shift}\ (\lambda k.\ 3\times(k\ 5))\ ) \tag{2.2}$$

In the code above, the operator `reset` first captures a delimited continuation (emphasized by the dark grey box) that does not spread to the whole rest of computation, and `shift` defines how the captured delimited continuation should be applied. Informally, one considers the "outer" continuation $1+\square$ and the delimited continuation $k = 2+\square$, captured by a control delimiter `reset`. According to the semantics of `shift` and `reset`, the captured delimited continuation will be applied as a function k in the body of the lambda-expression given as argument to `shift`, so expression (2.2) turns into the following one:

$$1+\ 3\times(\ 2+\ 5\ )$$

[9] A proper implementation of the example in Racket would look like `(+ 1 (reset (+ 2 (shift k (* 3 (k 5))))))`.

and evaluates to 22. It is worth mentioning that, unlike `call/cc`, the considered delimited control operators do not abort control flow and instead compose the captured continuation with the rest of the computation.

Recently, delimited control operators received a lot of attention because of their applications to the effective implementation of asynchronous computations [177], partial evaluation [13, 48], code generation [200] and mobile computing [195]. Also, possible formalizations of the delimited control operators with multiple prompts have been recently investigated [71] along with their behavioral theory [20].

We address the reader interested in intuition behind the pushy and jumpy continuations to the work by Flatt et al. [88]. The introduction of Dariusz Biernacki's dissertation [19] provides an exhaustive overview of control operators in higher-order languages; the same work also connects different styles of semantics for delimited continuations.

2.3 Transformations of Functional Programs

After making acquaintance with the main concepts of higher-order functional programming in Section 2.2, we obtained a *way to define* the semantics of computations in a declarative way. Now, we come to the last subject of our study: functional program transformations, which can serve as a *toolbox* to inter-derive computations.

In this section, we briefly enumerate a series of relevant functional program transformations that either contribute to the derivations described in the present work, or are implicitly belong to the functional correspondence tool-chain. All computations we consider in this section are *behaviour-preserving*, i.e., they do not change the result of the program. What they alter is computational properties, and we are going to exploit this fact further to inter-derive semantics. Most of the transformations discussed below have left inverses, which makes them (and, therefore, the result of the derivation) reversible.

2.3.1 CPS transformation

A *CPS transformation* is a function that takes a source program in a direct style and converts it into continuation-passing style. The original formulation of CPS transformation is due to Fischer [85] and Plotkin [165].

The original CPS transformation implemented in such a way that produces a program with a large amount of rudimentary reductions induced by passing continuations explicitly everywhere in the program, which are usually referred to as *administrative reductions*. Danvy and Filinski suggested an alternative transformation that yields a

resulting program with a minimal amount of administrative reductions [54]. Danvy has described a simple algorithm to transform lambda-terms into CPS that amounts to three steps [46]:

1. Give a name to each *intermediate* application.
2. Sequentialize the evaluation of these named applications by a *traversal* of their syntax tree. The tree traversal will therefore mimic the reduction strategy, for instance, the "innermost" expression will be put first.[10]
3. The resulting expression is equipped with a *continuation*.

For instance, the result of applying the 3-steps CPS transformation to the expression[11]

$$f\ (+\ (g\ 1)\ (h\ 2)) \tag{2.3}$$

is

$$\lambda k.\ \widehat{g}\ 1\ (\lambda x_1.\ \widehat{h}\ 2\ (\lambda x_2.\ \widehat{+}\ x_1\ x_2\ (\lambda x_3.\ \widehat{f}\ x_3\ k))) \tag{2.4}$$

where $\widehat{f}$, $\widehat{g}$, $\widehat{h}$ and $\widehat{+}$ are CPS-transformed versions of functions f, g, h and $+$, taking an extra parameter for a continuation, and k is the initial continuation.

2.3.2 Direct-style transformation

Direct-style transformation is a *left inverse* of CPS transformation in the sense that it translates a CPS-transformed program to the equivalent program in direct style. The algorithm of the direct-style transform can be easily obtained by inverting the three steps of the CPS transformation: continuations are turned to subsequent **`let`**-expressions, whose right-hand sides can be then inlined, which eliminates the need to pass the continuation as a parameter. The direct-style transform was originally formulated by Danvy [47], and later Danvy and Lawall studied the direct-style transformation extended to first-class continuations [57].

It is important to notice that in the presence of continuation dropping due to non-local returns, the direct-style transform might require the use of control operators such as `call/cc`.

[10] Which corresponds exactly to the administrative normal form [87].

[11] Here we consider the plus operation $+$ as a function of two parameters, so it is written in the prefix form.

2.3.3 Defunctionalization

Defunctionalization is a program transformation that converts a higher-order program with first-class functions into an equivalent first-order program. Defunctionalization was pioneered by Reynolds in his work on definitional interpreters [174] and later formulated algorithmically by Danvy and Nielsen and studied in a series of applications [62].

Defunctionalization is a global program transformation that changes the representation of a function space (i.e., all first-class functions in a program) into a first-order algebraic datatype, represented by a series of constructors. This data type enumerates all inhabitants of the function space. Each inhabitant is represented as a tagged summand (i.e., a particular datatype constructor) binding the values corresponding to free variables of the represented function, thereby capturing the *environment* component of the function (see the discussion about closures in Section 2.2.3). The *code* component of a function is described by providing a special *dispatcher* function, traditionally referred in the literature as `apply`. The dispatcher function is defined as a set of disjoint clauses, where each of the clauses corresponds to the body of a particular function space inhabitant.

From the practical perspective, defunctionalization can be considered as a simple technique to translate higher-order code to first-order, which makes numerous optimization techniques from the first-order world available in higher-order languages.

Multiple practical problems arise when one tries to implement defunctionalization efficiently. First, it seems like a heavy-weight solution to implement the *whole* function space of a program using just one algebraic datatype and one dispatcher function. The usual implementation technique involves computing of simple *flows-to* information of the program, determining statically which functions can be invoked at which call sites. This is normally done using a context-insensitive control-flow analysis [186, 138].[12] Another idea, usually referred to as *lightweight defunctionalization* consists of including in the list of variables bound by a particular summand of a datatype only those free variables that cannot be statically proven to refer to the same run-time entity (i.e., there might be more than one closure instance arising from the same lambda-abstraction) [14]. This idea allows one to substitute an indirect function call with a direct call to a statically known function, which can be pushed further by using the techniques to solve the generalized environment problem in higher-order languages [138, 140].

The essential property of defunctionalization, which we are going to exploit in forthcoming chapters, is that it turns a composed continuations in the form of anonymous

[12] Traditionally, the simplest possible *context-insensitive* control-flow analysis is used to split the function space into disjoint components prior to performing defunctionalization. The impact of using a *context-sensitive* control-flow analysis (also known as *k*-CFA) for performing the defunctionalization is still unclear.

lambda-abstraction into a stack-like structure in a zipper-like fashion [105], thereby bridging the evaluators in CPS and stack-based transition systems.

2.3.4 Refunctionalization

Refunctionalization is a left inverse of defunctionalization. While defunctionalization converts a higher-order program into a first-order program by replacing first-class functions by constructors of an appropriate datatype and supplying a dispatcher function, refunctionalization translates a first-order program into an equivalent higher-order program by substituting constructors of the defunctionalized datatype and inlining the corresponding clauses of the dispatcher function, turning them into the bodies of anonymous functions [50]. Refunctionalization has been demonstrated to be useful for proving the equivalence of data-processing programs in the functional accumulator-passing style and the continuation-passing style (e.g., the `reverse` function on lists). Moreover, it turns out that most instances of the Zipper [105] are defunctionalized continuations.

In the story of this dissertation, refunctionalization will play a crucial role for the derivation of a recursive descent type checker from the reduction type checking semantics.

2.3.5 Deforestation

Deforestation is a program transformation that eliminates intermediate construction of tree-like structures when processing them. The term and the technique are originally introduced by Philip Wadler [206]. Sometimes, deforestation is defined in terms of a *hylomorphism*, the composition of an *anamorphism* (i.e., a construction of a data structure) and a *catamorphism* (i.e., a compositional traversal of a structure, also known as *fold*) [133]. Once a composition of functions is proven to be a hylomorphism, it is a valid target for deforestation.

Lists are a particular case of trees, and, as we will show in Chapter 3, reduction contexts form a list-like structure. In the study of the application of functional correspondence to program semantics, deforestation occurs when transforming an implementation of a reduction semantics to avoid subsequent decomposition and recomposition of the reduction contexts. We will use deforestation as the first derivation step in Chapter 3.

2.3.6 Lightweight fusion

Fusion is a general term for the program optimization technique that involves the combination of composition of adjacent computation into *one* computation. A particular case of fusion is *deforestation* (see Section 2.3.5). Ohori and Sasano suggested a method referred to as *lightweight fusion* [158] that works for general recursive functions on general algebraic data types. The essence of the method is in extending the function inlining process with a new *fusion law* that translates a term of the form $(\texttt{fix}\ g.\lambda x.e) \circ f$ into a new fixed point term $\texttt{fix}\ h.(\lambda x.e')$ by promoting the function f through the fixed point operator resulting in a particular expression e' in the body of `fix`.

An example of such a fixed point fusion is the derivation of a big-step abstract machine from a small-step abstract machine—a state-transition function together with a driver loop in a trampolined style (see Section 2.3.7). The initial small-step abstract machine is represented by a transition function f and a "driver loop" g, whose fixed point is computed [59]. The resulting big-step machine is represented by a recursive function that maps an initial state to a terminal result.

The lightweight fusion approach has some significant limitations. For instance, one cannot fuse two successive applications of the same function or apply the fusion transformation to mutually recursive functions.

2.3.7 Trampoline style and trampoline transform

Trampoline style is a programming technique that is used in some of the Lisp implementation as an optimization to turn tail-recursive or mutually-recursive functions into the form of a small-step abstract machine [91]. Trampoline style-programming is implemented as a library function in the Lisp dialect known as Clojure [101] as a way to overcome the lack of tail-call optimization in the Java Virtual Machine [173].

The trampolined recursive function is turned into a driver loop such that each iteration returns a *thunked* value for the "next step". I.e., instead of actually making a call in its tail, a trampolined function returns the function to be called (a *thunk*), and the driver loop calls it, thus allowing stepwise execution without stack growth. In the forthcoming chapters we will demonstrate the implicit use of the trampoline transform technique by subsequent CPS transformation and defunctionalization. The resulting program will exhibit tail-recursive behaviour, which makes it the subject of a straightforward driver loop extraction. Abusing terminology, the trampoline transform can be considered as an inverse of lightweight fusion (see Section 2.3.6). i.e., so-called *lightweight fission*.

2.3.8 Lambda lifting

Lambda lifting is a program transformation that names all lambda-abstractions in a functional program and turns them into global functions by moving them to the program top level [114]. To be performed correctly, lambda lifting is almost always preceded by closure conversion, so these two notions are often confused in the compiler literature. Lambda lifting has been formally specified and proved to be correct by Fischbach and Hannan [84]. Later, Danvy and Schultz implemented an quadratic-time algorithm [65], in contrast with the original cubic-time translation. In object-oriented community, where anonymous classes play the role of lambda-abstractions, closure conversion and lambda-lifting might be thought of as the refactorings "convert anonymous class to inner" and "move inner class to upper level". These refactorings are implemented, for instance, in the IntelliJ IDEA programming environment [110].

2.3.9 Closure conversion

Closure conversion is a global program transformation that eliminates free variables in lambda abstractions and inner functions by passing an explicit environment parameter instead. The result of the transformation is a program that can be a subject of lambda lifting. Closure conversion is different from defunctionalization in the sense that it neither introduces a data type to represents inhabitants of the function space, nor replaces lambda-abstractions with constructors of this datatype. However, closure conversion is similar to defunctionalization in the sense that it performs the analysis of free variables in the bodies of lambda-abstractions, before extracting them as parameters. Closure conversion is *different* from lambda lifting since it just eliminates free variables, but does not turn inner and functions and lambda-abstractions into top-level functions. However, closure conversion is almost always a transformation preceding lambda lifting. The resulting program can incur a quadratic blowup in size because of additional variable passing. Techniques, similar to those used in lightweight defunctionalization [14] can be used in order to reduce the amount of free variables extracted as parameters [190, 194].

Together with lambda lifting, closure conversion have application as a tool in the functional correspondence chain providing the way to mechanically connect denotational semantics with natural (big-step) semantics of an arbitrary computational formalism [174]. Although remarkable, we leave this aspect uncovered in the present work.

2.3.10 Lambda dropping

Lambda dropping is a left inverse from lambda lifting. The technique was originally proposed by Danvy and Schultz [64] as a way to restore the original lexical block structure of a functional program with recursive equations. Strictly speaking, lambda dropping is an inverse of the composition of lambda lifting with closure conversion: the essence of the transformation is that it eliminates the parameters that are always used in the same scope, so the function does not need to pass them along: they are instead replaced by local variables. In this respect, the goal of the lambda dropping is coherent with dependency analysis as it is described by Peyton Jones [162], which is used to generate block structure from recursive equations, relying on a similar algorithm referred to as "block sinking". Danvy and Schultz also show how lambda dropping can be applied for partial evaluation and revealing global read-only variables by localizing lexical blocks.

2.3.11 Contification

Contification is a compiler optimization that turns a function that always returns to the *same* program point into a continuation [89] of its callers. Indeed, if a function always returns to the same program point, then this function's calls and returns can be viewed as describing *intra*procedural instead of *inter*procedural control-flow. In a sense, contification is similar to lambda-dropping, as it also moves functions from the global program level to bodies of other functions. Contification is used as an intermediate program transformation in optimizing compilers of functional programming languages such as MLton [208], after the program has been turned into first order by closure conversion. The functions to be contified are usually determined using dominance analysis on a static program call graph. From a functional correspondence perspective, contification can be considered as a technique to extract *inner* subroutines of an evaluator, for example, extracting contraction functions.

2.3.12 Other transformations

In our study, we will also make use of a series of "small" transformations, although used in production compilers, are not referred to by any specific name due to their simplicity. We will assign some names to them locally to this thesis.

Result-stack extraction By *result stack extraction* we mean replacing local variables of a function by a global stack containing the results of intermediate calls.

$$\text{(fib-1)}\ \frac{}{1 \Downarrow_{fib} 1} \qquad \text{(fib-2)}\ \frac{}{2 \Downarrow_{fib} 1}$$

$$\text{(fib-n)}\ \frac{(n-1) \Downarrow_{fib} v_1 \qquad (n-2) \Downarrow_{fib} v_2}{n \Downarrow_{fib} v_1 + v_2}$$

Figure 2.7: Natural semantics of Fibonacci numbers computation

The result stack component is passed along function calls as an extra parameter. It stores intermediate values after they have been computed but before they are used. Evaluating an expression leaves its result on top of the data stack. Function calls, therefore, expect to find their argument and the to-be-called function on top of this data stack. In the case of nested calls, the immutable part of the stack is saved by the caller, whereas, a callee is invoked with a reduced or fresh stack.

Control-stack extraction As has been pointed out, the result of defunctionalizing a program in CPS is a data type representing continuations that typically has a stack structure, reminiscent of the Zipper or continuations of a CEK machine. In our transformations, we make this structure explicit by refactoring the result of defunctionalization into a stack and passing it along to function calls as an extra argument.

2.4 Pulling it All Together: Inter-Deriving Semantics for Fibonacci Numbers

In the concluding section of this chapter we put all presented components together and demonstrate the interplay between different semantics and interpreters on a toy example—Fibonacci numbers.

Big-step semantics for Fibonacci numbers

When presenting Fibonacci numbers, one usually does it in the form of a recurrent equation. Such an equation can be presented as a "semantics", whose inputs are just natural numbers (Figure 2.7). Armed with the knowledge about operational semantics

from Section 2.1.3, the reader can easily recognize that this definition is expressed using the big-step operational formalism.[13]

Our goal is to provide a small-step stack-based register abstract machine to compute Fibonacci numbers. In the subsequent sections we show how to do this by employing functional correspondence. At each stage of the transformation we increase the index of the function `fib`. Parts of the code essential for each derivation are highlighted by grey boxes.

Initial implementation

Our first step is to implement the formalism from Figure 2.7 in the form of a recursive evaluator in Standard ML, which brings us to the following definition.

```
fun fib0 n
  = if n = 1 orelse n = 2 then 1
    else let val v1 = fib0 (n - 1)
             val v2 = fib0 (n - 2)
         in v1 + v2 end
```

Extracting a result stack

We rewrite the procedure `fib0` in a way that fixes the order between the computed intermediate results `v1` and `v2` by replacing local variables by a result stack passed as an extra parameter.

```
fun fib_stack (s: int list, n: int)
  = if n = 1 orelse n = 2 then 1 :: s
    else let val s1 = fib_stack (s, n - 1)
             val s2 = fib_stack (s1, n - 2)
             in case s2 of
                 v1 :: v2 :: s3 => (v1 + v2) :: s3
             end

fun fib1 n = fib_stack (nil, n)
```

The result is a callee-save, explicit stack-threading evaluator [58].

[13] The first Fibonacci number is computed for 1 and not for 0: it was unconsciously picked by the author according to the tradition of the Russian mathematical school, where natural numbers start from one. This choice became conscious during the discussion with Olivier Danvy at the preliminary defense of the thesis.

CPS transformation

Our evaluator `fib_stack` has a number of non-tail calls to itself:

```
fun fib_stack (s: int list, n: int)
  = if n = 1 orelse n = 2 then 1 :: s
    else let val s1 = fib_stack (s, n - 1)
             val s2 = fib_stack (s1, n - 2)
             in case s2 of
                v1 :: v2 :: s3 => (v1 + v2) :: s3
             end

fun fib1 n = fib_stack (nil, n)
```

We turn them into tail calls by employing the CPS transformation (see Section 2.3.1). The CPS-transformed evaluator now looks as follows:

```
fun fib_cps (s, n, k)
  = if n = 1 orelse n = 2 then k (1 :: s)
    else fib_cps (s, n - 1, fn s1 =>
           fib_cps (s1, n - 2, fn s2 =>
             case s2 of
               v1 :: v2 :: s3 => k ((v1 + v2) :: s3)))

fun fib2 n = fib_cps (nil, n, fn (x :: _) => x )
```

Defunctionalization

After we have CPS-transformed our evaluator, we see a number of anonymous functions representing continuations. We turn these continuations into a first-order datatype by employing defunctionalization (see Section 2.3.3):

```
datatype cont = CONT_MT
              | CONT_FIB1 of int * cont
              | CONT_FIB2 of cont

fun fib_defun (s, n, C)
  = if n = 1 orelse n = 2 then continue (1 :: s, C)
    else fib_defun (s, n - 1, CONT_FIB1 (n, C) )

and continue (s, CONT_MT )
    = (case s of (x :: _) => x)
  | continue (s, CONT_FIB1 (n, C) )
    = fib_defun (s, n - 2, CONT_FIB2 C )
  | continue (s, CONT_FIB2 C )
    = case s of (v1 :: v2 :: s3) => continue ((v1 + v2) :: s3, C)

fun fib3 n = fib_defun (nil, n, CONT_MT )
```

Unifying control

The result of defunctionalization is used by the function `continue` to dispatch calls. The *number* component is used for dispatch by the function `fib_defun`. Our next transformation is *control unification*: we provide a function `fib_defun'` and a datatype `cont'` that implement dispatch on both the structure of integer and the structure of a continuation:

```
datatype cont' = CONT_MT'
               | CONT_FIB1' of int * cont'
               | CONT_FIB2' of cont'
               | NUM' of int * cont'

fun fib_defun' (s, NUM' (n, C) )
  = if n = 1 orelse n = 2 then continue1 (1 :: s, C)
    else fib_defun' (s, NUM' (n - 1, CONT_FIB1' (n, C)))

and continue1 (s, CONT_MT' )
    = (case s of (x :: _) => x)
  | continue1 (s, CONT_FIB1' (n, C) )
    = fib_defun' (s, NUM' (n - 2, CONT_FIB2' C))
  | continue1 (s, CONT_FIB2' C )
    = case s of (v1 :: v2 :: s3) => continue1 ((v1 + v2) :: s3, C)

fun fib4 n = fib_defun' (nil, NUM' (n, CONT_MT'))
```

Introducing a control stack

The enhanced continuation data type `cont'` has a list-like structure with three constructors taking arguments and `CONT_MP` playing the role of `nil`. Let's turn it into a traditional ML-style list by introducing a dedicated data type `control_element` for control elements. The `cont'` data type then turns into a *control stack*, which is depicted by the following implementation.

```
datatype control_element = NUM of int
                         | CF1 of int
                         | CF2

fun fib_control (s, NUM n :: C)
  = if n = 1 orelse n = 2 then fib_control (1 :: s, C)
    else fib_control(s, NUM (n - 1) :: CF1 n :: C)
  | fib_control (s, CF1 n :: C)
    = fib_control (s, NUM (n - 2) :: CF2 :: C)
  | fib_control (s, CF2 :: C)
    = (case s of (v1 :: v2 :: s3) => fib_control ((v1 + v2) :: s3, C))
  | fib_control (s, nil)
    = (case s of (x :: _) => x)

fun fib5 n = fib_control (nil, NUM n :: nil)
```

From a big-step to a small-step abstract machine

One can notice that the function `fib_control` in the previous section is tail-recursive, i.e., all the calls it performs are tail calls to itself. Such a function is a candidate for a lightweight fission, which yields the transition function `step` and driver loop `iterate`.

$$
\begin{array}{rcl}
\langle S,\ Num(1) :: C\rangle & \Rightarrow_{\mathrm{SC}_{fib}} & \langle 1 :: S,\ C\rangle \\
\langle S,\ Num(2) :: C\rangle & \Rightarrow_{\mathrm{SC}_{fib}} & \langle 1 :: S,\ C\rangle \\
\langle S,\ Num(n) :: C\rangle & \Rightarrow_{\mathrm{SC}_{fib}} & \langle S,\ Num(n-1) :: CF_1(n) :: C\rangle \\
\langle S,\ CF_1(n) :: C\rangle & \Rightarrow_{\mathrm{SC}_{fib}} & \langle S,\ Num(n-2) :: CF_2 :: C\rangle \\
\langle v_1 :: v_2 :: S,\ CF_2 :: C\rangle & \Rightarrow_{\mathrm{SC}_{fib}} & \langle (v_1 + v_2) :: S,\ C\rangle
\end{array}
$$

Figure 2.8: A small-step abstract machine for Fibonacci numbers

```
type state = int list * control_element list

(*  step : state -> state  *)
fun step ( s, NUM 1 ::  C )
    = (1 :: s, C)
  | step ( s, NUM 2 ::  C )
    = (1 :: s, C)
  | step ( s, NUM n ::  C )
    = (s, NUM (n - 1) :: CF1 n :: C)
  | step (s,  CF1 n ::  C )
    = (s, NUM (n - 2) :: CF2 :: C)
  | step ( v1 ::  v2 ::  s3, CF2 ::  C )
    = ((v1 + v2) :: s3,  C)

(*  step : state -> int  *)
fun iterate (v :: _, nil)
    = v
  | iterate (s, C)
    = iterate (step (s, C))

(*  fib6 : int -> int  *)
fun fib6 n = iterate (nil, NUM n :: nil)
```

Based on the correspondence described in Section 2.1.5 and a shape of the function `step`, we can easily extract the definition of a small-step operational semantics, corresponding to an abstract machine with a two-component state. The formal descriptions of this machine is presented in Figure 2.8. The machine is reminiscent of Landin's SECD machine with only two components: a result stack S and a control stack C. This formal definition ends our derivation.

Chapter 3

From Type Checking via Reduction to Type Checking via Evaluation

This chapter is the first part of the story about mechanical inter-derivation of type checking semantics.

We connect two independently investigated formalisms: one, due to Kuan et al., in the form of a term rewriting system and the other in the form of a traditional set of derivation rules, thus, initiating the left part of the diagram in Figure 1.4. By employing a set of techniques investigated by Danvy et al. [4, 51, 58, 59, 60, 62], we mechanically derive the correspondence between a *reduction-based* semantics for type-checking and a traditional one in the form of derivation rules, implemented as a *recursive descent*. The correspondence is established through a series of semantics-preserving functional program transformations.

Following the presented methodology of semantic correspondence, we connect different semantics of type checking by the construction and inter-derivation of their computational counterparts. Thus, *no* soundness and completeness theorem need to be proven: they are instead corollaries of the correctness of inter-derivation and of the initial specification. Starting from the implementation of a reduction-based semantics, we employ a series of semantics-preserving functional-program transformations to eventually obtain a traditional recursive descent for type-checking. The transformations we use are off-the-shelf [52], and we invite an interested reader to take a look on the overview of the available techniques with references to the corresponding correctness proofs [51].

$$\text{(t-var)}\frac{(x:\tau)\in\Gamma}{\Gamma\vdash x:\tau} \qquad \text{(t-lam)}\frac{\Gamma,\, x:\tau_1\vdash e:\tau_2}{\Gamma\vdash \lambda x:\tau_1.e:\tau_1\to\tau_2}$$

$$\text{(t-app)}\frac{\Gamma\vdash e_1:\tau_1\to\tau_2 \quad \Gamma\vdash e_2:\tau_1}{\Gamma\vdash e_1\ e_2:\tau_2} \qquad \text{(t-num)}\frac{}{\Gamma\vdash \mathit{number}:\mathsf{num}}$$

Figure 3.1: Church-style type checking rules of the simply typed lambda calculus, revisited.

3.1 Starting Point: a Hybrid Language for Type Checking

We consider a reduction system for type checking the simply typed lambda calculus. The system was originally proposed by Kuan et al. [123] and is presented as a case study in the scope of PLT Redex framework [122]. The approach scales to Curry-Hindley type inference and Hindley-Milner let-polymorphism. The techniques presented in the current work can be adjusted to these cases by adding unification variables, so for the sake of brevity we examine only the simplest model. The hybrid language $\lambda_{\mathcal{H}}$ and its semantics are described in Figure 3.2. The reduction system introduces a type-checking context T that induces a left-most, inner-most order of reduction. Variable occurrences are replaced by their types at the moment a λ-abstraction is reduced, according to rule [tc-lam]. Rule [tc-lam] also introduces the arrow type constructor. Finally, rule [tc-$\tau\beta$] syntactically matches the function parameter type against an argument type.

The classical way to represent type checking is via a collection of logical derivation rules assuming a construction of a proof-tree for a well-typed program. Such rules for the simply typed lambda calculus are given in Figure 3.1. According to Kuan et al., a complete type reduction sequence is one that reduces to a type. The existence of a complete reduction sequence resulting by a type is the case for well-typed terms only. The following theorem states that a complete type reduction sequence corresponds to a complete type derivation proof tree for a well-typed term in the host language and vice versa.

Theorem 3.1.1. [123] (Soundness and Completeness for $\mapsto_t$)

For any e and τ, $\emptyset\vdash e:\tau$ *iff* $e\mapsto_t^*\tau$

The question we address in this chapter is whether a natural correspondence between

Hybrid terms	e	$::=$	$n \mid x \mid \lambda x : \tau.e \mid e\ e \mid \tau \to e \mid \mathsf{num}$
Numbers	n	$::=$	*number*
Types	τ	$::=$	$\mathsf{num} \mid \tau \to \tau$
Type-checking contexts	T	$::=$	$T\ e \mid \tau\ T \mid \tau \to T \mid [\,]$

Hybrid language and type-checking contexts

$$
\begin{array}{rcll}
T[n] & \mapsto_t & T[\mathsf{num}] & [\text{tc-const}] \\
T[\lambda x : \tau.e] & \mapsto_t & T[\tau \to \{\tau/x\}\ e] & [\text{tc-lam}] \\
T[(\tau_1 \to \tau_2)\ \tau_1] & \mapsto_t & T[\tau_2] & [\text{tc-}\tau\beta]
\end{array}
$$

Type-checking reduction rules

Figure 3.2: Reduction semantics of $\lambda_{\mathcal{H}}$.

these semantics exists which avoids the need for the soundness and completeness theorems. The answer to this question is positive and below we show how to derive a traditional type-checker mechanically from the given rewriting system.

3.1.1 Chapter outline

The remainder of the chapter is structured as follows. Section 3.2 gives an overview of our method, enumerating the techniques involved. Section 3.3 describes an implementation of the hybrid language and its reduction semantics in Standard ML. Section 3.4 describes a set of program transformations corresponding to the transition from the reduction-based semantics for type inference to a traditional recursive descent.

3.2 Method Overview

An overview of the program transformations is shown in Figure 4.2. We start by providing the implementation of a hybrid language for the simply typed lambda calculus, a notion of closures in it and a corresponding reduction semantics via contraction as a starting point for further transformations (Section 3.3). The reduction-based normalization function is transformed to a family of reduction-free normalization functions, i.e., ones where no intermediate closure is ever constructed. In order to do so, we first refocus the reduction-based normalization function to

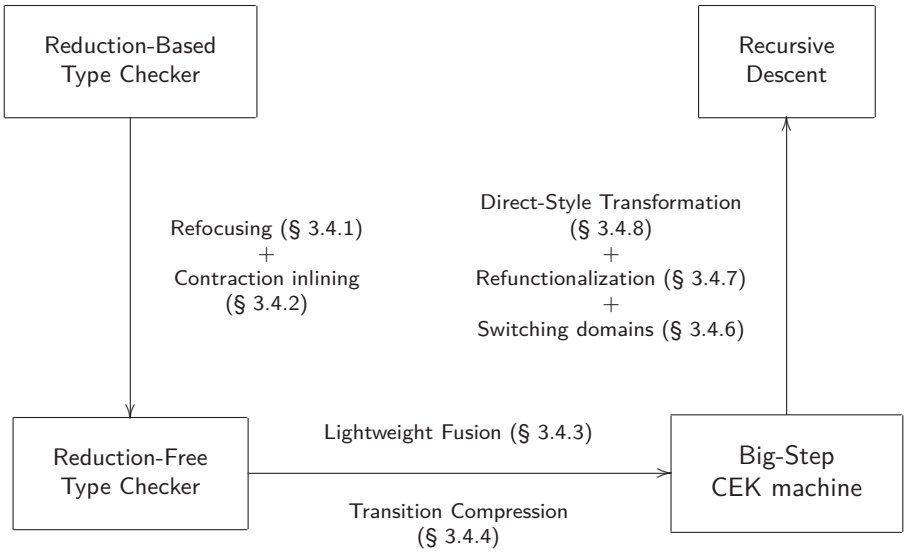

Figure 3.3: Inter-derivation from a reduction-based to a compositionsl type checker

obtain a small-step abstract machine implementing the iteration of the refocus function (Section 3.4.1). After inlining the contraction function (Section 3.4.2), we transform this small-step abstract machine into a big-step one by applying a technique known as "lightweight fusion by fixed-point promotion" [59] (Section 3.4.3). This machine exhibits a number of corridor transitions, which we then compress (Section 3.4.4). We then flatten its configurations and rename its transition functions to something more intuitive (Section 3.4.5). We also switch domains of evaluator functions to factor out artifacts of the hybrid language (Section 3.4.6). The resulting abstract machine is in defunctionalized form, so we refunctionalize it (Section 3.4.7). The result is in continuation-passing style, so we transform it into direct style (Section 3.4.8). The final result is a traditional compositional type-checker.

In Section 3.4.8 we rely on the library of undelimited continuations to model top-level exceptions. For the sake of brevity, we omit some program artifacts (sometimes only giving their signature), keeping only essential parts to demonstrate the corresponding program transformation. The reader interested in particular details of the implementation of helper functions is welcome to take a look the accompanying code overview in Appendix B. All essential definitions of functions and datatypes in the text can be found using the index at the end of the dissertation. At each transformation stage the trailing index of all involved functions is incremented.

3.3 A Reduction-Based Type Checker

This section provides the initial implementation of $\lambda_{\mathcal{H}}$ in SML, which will be used for further transformations in Section 3.4.

3.3.1 Reduction-based hybrid term normalization

The reduction-based normalization of hybrid terms is implemented by providing an abstract syntax, a notion of contraction and a reduction strategy. Then we provide a one-step reduction function that decomposes a non-value closure into a potential redex and a reduction context, contracts the potential redex, if it is actually one, and then recomposes the context with the contractum. Finally we define a reduction-based normalization function that repeatedly applies the one-step reduction function until a value (i.e., an actual type of an expression) is reached.

In the specification of $\lambda_{\mathcal{H}}$, the contraction of lambda expressions (rule [tc-lam]) is specified using a meta-level notion of capture-avoiding substitutions. However, most implementations do not use actual substitutions and keep an *explicit representation* of what should be substituted on demand, leaving the term untouched [79, pages 100–105]. To model *explicit substitutions*, we chose the applicative order version of Curien's calculus, which uses closures, i.e, terms together with their lexical environment [17]. The cited paper also relates values in the language of closures with values in λ-calculus (see Section 2.5). From the implementation perspective, it is done by introduction of embedding functions. The environments map variables to values (i.e., types in this case) while reducing an expression, which corresponds do the capture-avoiding substitution strategy [52, Section 6]. The chosen calculus allows us to come eventually in Section 3.4 to a well-known representation of a type-checking algorithm with an environment Γ, which predictably serves the same purpose, i.e., mapping variables to their types.

3.3.2 Abstract syntax of $\lambda_{\mathcal{H}}$: closures and values

The abstract syntax for $\lambda_{\mathcal{H}}$, which is presented in Figure 3.2, is described in SML below. It includes integer literals, identifiers, lambda-abstractions, applications as well as hybrid elements such as numeric types and arrows $\tau \to e$. Types are either numeric types or arrow types. The special value `T_ERROR s` is used for typing errors; it cannot be a constituent of any other type. A value in the hybrid language is either an integer or a function type.

```
datatype typ = T_NUM
             | T_ARR of typ * typ
             | T_ERROR of string

datatype term = LIT of int
              | IDE of string
              | LAM of string * typ * term
              | APP of term * term

datatype hterm = H_LIT of int
               | H_IDE of string
               | H_LAM of string * typ * hterm
               | H_APP of hterm * hterm
               | H_TARR of typ * hterm
               | H_TNUM
```

Typing environments `TEnv` represent bindings of identifiers to types, which are values in the hybrid language. In order to keep to the uniform approach for different semantics for type inference [181], we leave environments parametrized by the type parameter `'a`, which is instantiated with `typ` in this case.

```
signature TEnv = sig
  type 'a gamma
  val empty :  (string * 'a) gamma
  val extend : string * 'a * (string * 'a) gamma -> (string * 'b) gamma
  val lookup : string * (string * 'a) gamma -> 'a option
end
```

We introduce closures into the hybrid language in order to represent the environment-based reduction system. A closure can either be a number, a ground closure pairing a term and an environment, a combination of closures, a closure for a hybrid arrow expression, or a closure for a value arrow element, namely an arrow type. Environments bind identifiers to values.

```
datatype closure = CLO_NUM
                 | CLO_GND of hterm * bindings
                 | CLO_APP of closure * closure
                 | CLO_ARR of typ * closure
                 | CLO_ARR_TYPE of typ
withtype bindings = typ TEnv.gamma
```

We also specify the corresponding embeddings of values to closures and of terms to hybrid terms (the definitions are omitted and can be found in Appendix B):

```
val type_to_closure : typ -> closure
val term_to_hterm : term -> hterm
```

3.3.3 Notion of contraction

A potential redex is either a numeric literal, a ground closure pairing an identifier and an environment, the application of a value to another value, a lambda-abstraction to be type-reduced, an arrow type, or a ground closure pairing a term application and an environment. All these possibilities are described by the following data type:

```
datatype potential_redex
  = PR_NUM
  | PR_IDE of string * bindings
  | PR_APP of typ * typ
  | PR_LAM of string * typ * hterm * bindings
  | PR_ARR of typ * typ
  | PR_PROP of hterm * hterm * bindings
```

A potential redex may trigger a contraction or it may get stuck. These outcomes are captured by the following datatype:

```
datatype contractum_or_error = CONTRACTUM of closure
                             | ERROR of string
```

The string content of `ERROR` is an error message.

The contraction function `contract` reflects the type-checking reduction rules for $\lambda_{\mathcal{H}}$. For instance, any integer literal contracts to a numeric type `T_NUM`, a lambda expression contracts to an arrow expression of the hybrid language, and the contraction of a potential redex `PR_APP` checks whether its first parameter is a function type and its parameter type matches the argument of the application, and then, if it is the case, replaces the application by a function result type.

```
(*  contract: potential_redex -> contractum_or_error  *)
fun contract PR_NUM
    = CONTRACTUM CLO_NUM
  | contract (PR_ARR (t1, t2))
    = CONTRACTUM (type_to_closure (T_ARR (t1, t2)))
  | contract (PR_IDE (x, bs))
    = (case TEnv.lookup (x, bs)
        of NONE     => ERROR "undeclared identifier"
         | (SOME v) => CONTRACTUM (type_to_closure v))
  | contract (PR_LAM (x, t, e, bs))
    = CONTRACTUM (CLO_GND (H_TARR (t, e), TEnv.extend (x, t, bs)))
  | contract (PR_APP (T_ARR (t1, t2), v))
    = if t1 = v
      then CONTRACTUM (type_to_closure t2)
      else ERROR "parameter type mismatch"
  | contract (PR_PROP (t0, t1, bs))
    = CONTRACTUM (CLO_APP (CLO_GND (t0, bs), CLO_GND (t1, bs)))
  | contract (PR_APP (t1, t2))
```

```
    = ERROR "non-function application"
```

A non-value closure is stuck when an identifier does not occur in the current environment or non-function type is used in a function position or a function parameter's type does not correspond to the actual argument's type.

3.3.4 Reduction strategy

Reduction contexts are defined as follows:

```
datatype hctx = CTX_MT
              | CTX_FUN of hctx * closure
              | CTX_ARG of typ * hctx
              | CTX_ARR of typ * hctx
```

A context is a closure with a hole, represented inside-out in a zipper-like fashion [105]. Following the description of $\lambda_{\mathcal{H}}$'s reduction semantics we seek the left-most inner-most potential redex in a closure. In order to reduce a closure, it is first decomposed. The closure might be a value and not contain any potential redex, otherwise it can be decomposed into a potential redex and a reduction context. These possibilities are captured by the following datatype:

```
datatype type_or_decomposition = VAL of typ
                               | DEC of potential_redex * hctx
```

A decomposition function recursively searches for the left-most inner-most redex in a closure. Examples of some specific decomposition functions may be found in recent work of Danvy [52]. In our implementation we define decomposition (`decompose`) as a big-step abstract machine with two state-transition functions, `decompose_closure` and `decompose_context`. The former traverses a given closure and accumulates the reduction context until it finds a value and the latter dispatches over the accumulated context to determine whether the given closure is a value or a potential redex. The function `decompose` starts by decomposing a closure within an empty context. For the full definition of the decomposition functions, see the accompanying code. The recomposition function `recompose` takes a context and a value to embed, peels off context layers and iteratively constructs the resulting closure. The implementation of these functions is essential for the further derivation, so we provide it below:

```
(*  decompose_closure : closure * hctx -> type_or_decomposition *)
fun decompose_closure (CLO_NUM, C)
    = decompose_context (C, T_NUM)
  | decompose_closure (CLO_ARR_TYPE v, C)
    = decompose_context (C, v)
  | decompose_closure (CLO_GND (H_LIT n, bs), C)
    = decompose_context (C, T_NUM)
  | decompose_closure (CLO_GND (H_IDE x, bs), C)
    = DEC (PR_IDE (x, bs), C)
  | decompose_closure (CLO_GND (H_LAM (x, t, e), bs), C)
    = DEC (PR_LAM (x, t, e, bs), C)
  | decompose_closure (CLO_GND (H_APP (t0, t1), bs), C)
    = DEC (PR_PROP (t0, t1, bs), C)
  | decompose_closure (CLO_GND (H_TNUM, bs), C)
    = decompose_context (C, T_NUM)
  | decompose_closure (CLO_GND (H_TARR (t, e), bs), C)
    = decompose_closure (CLO_GND (e, bs),
                         CTX_ARR (t, C))
  | decompose_closure (CLO_APP (c0, c1), C)
    = decompose_closure (c0, CTX_FUN (C, c1))
  | decompose_closure (CLO_ARR (v, c), C)
    = decompose_closure (c, CTX_ARR (v, C))

(*  decompose_context : hctx * typ -> type_or_decomposition *)
and decompose_context (CTX_MT, v)
    = VAL v
  | decompose_context (CTX_FUN (C, c1), v0)
    = decompose_closure (c1, CTX_ARG (v0, C))
  | decompose_context (CTX_ARG (v0, C), v1)
    = DEC (PR_APP (v0, v1), C)
  | decompose_context (CTX_ARR (v0, C), v1)
    = DEC (PR_ARR (v0, v1), C)

(*  decompose : closure -> type_or_decomposition *)
fun decompose c
  = decompose_closure (c, CTX_MT)

(*  recompose : hctx * closure -> closure *)
fun recompose (CTX_MT, c)
    = c
  | recompose (CTX_FUN (C, c1), c0)
    = recompose (C, CLO_APP (c0, c1))
  | recompose (CTX_ARG (v0, C), c1)
    = recompose (C, CLO_APP (type_to_closure v0, c1))
  | recompose (CTX_ARR (v0, C), c1)
    = recompose (C, CLO_ARR (v0, c1))
```

3.3.5 Reduction-based normalization

Reduction-based normalization is based on a function that iterates a one-step reduction function until it yields a value. At each iteration the normalization function inspects its argument. If it is a potential redex within some context it will be contracted using the function `contract` from Section 3.3.3 and then be recomposed. If during contraction an error occurs, it must be reported:

```
datatype result = RESULT of typ
                | WRONG of string

(*  iterate: type_or_decomposition -> result -> result  *)
fun iterate (VAL v) = RESULT v
  | iterate (DEC (pr, C))
    = (case contract pr
        of (CONTRACTUM c') => iterate (decompose (recompose (C, c')))
         | (ERROR s)       => WRONG s)
```

At this point we should take into account the fact the terms we want to type-check via reduction-based normalization are from the host language (and described by the data type `term`), whereas intermediate values of reductions are within the larger hybrid language (i.e., they are of type `hterm`). So we should first embed plain terms into hybrid ones using the function `term_to_hterm`.

The function `type_check` runs the reduction-based normalization function `normalize` and processes an obtained result.

```
(*  normalize: term -> result  *)
fun normalize t = iterate (decompose (CLO_GND (term_to_hterm t,
                                               TEnv.empty)))

(*  type_check: term -> typ  *)
fun type_check t
    = case normalize t
       of (RESULT v) => v
        | WRONG s    => T_ERROR s
```

3.4 From Reduction-Based to Compositional Type Checker

In this section, we follow a systematic approach to the construction of a reduction-free normalization function out of a reduction-based normalization function [52].

3.4.1 Refocusing

The operation of decomposing and recomposing a term is usually referred to as refocusing. As it has been pointed out by Danvy and Nielsen [63], a refocusing function may be expressed via the `decompose_closure` function, mentioned in Section 3.3.

```
(*  refocus : closure * hctx -> type_or_decomposition  *)
fun refocus (c, C) = decompose_closure (c, C)
```

The new version of the type checker differs from the original one by the definition of the function `iterate1` using the function `refocus` instead the composition of `decompose` and `recompose`. The type checker is now reduction-free, since no step-based reduction function is involved. The function `iterate1` operates directly on the result of refocusing instead.

```
(*  iterate1 : type_or_decomposition -> result  *)
fun iterate1 (VAL v)
    = RESULT v
  | iterate1 (DEC (pr, C))
    = (case contract pr
        of (CONTRACTUM c')
           => iterate1 (refocus (c', C))
         | (ERROR s)
           => WRONG s)

(*  normalize1 : term -> result  *)
fun normalize1 t
    = iterate1 (refocus (CLO_GND (term_to_hterm t,
                                  TEnv.empty), CTX_MT))

fun type_check t
    = case normalize1 t
       of (RESULT v) => v
        | WRONG s => T_ERROR s
```

3.4.2 Inlining the contraction function

We inline the function `contract` (Section 3.3.3) in the definition of `iterate1`. There are six cases in the definition of `contract`, so the `DEC` clause in the definition of `iterate1` is replaced by six `DEC` clauses. The resulting function is called `iterate2`.

```
fun iterate2 (VAL v)
    = RESULT v
  | iterate2 (DEC (PR_NUM, C))
    = iterate2 (refocus (CLO_NUM, C))
  | iterate2 (DEC (PR_ARR (t1, t2), C))
```

```
    = iterate2 (refocus (type_to_closure
                         (T_ARR (t1, t2)), C))
  | iterate2 (DEC (PR_IDE (x, bs), C))
    = (case TEnv.lookup (x, bs)
        of NONE
           => WRONG "undeclared identifier"
         | (SOME v) =>
           iterate2 (refocus (type_to_closure v, C)))
  | iterate2 (DEC (PR_LAM (x, t, e, bs), C))
    = iterate2 (refocus
                  (CLO_GND (H_TARR (t, e),
                      TEnv.extend (x, t, bs)), C))
  | iterate2 (DEC (PR_APP (T_ARR (t1, t2), v), C))
    = if t1 = v
      then iterate2 (refocus (type_to_closure t2, C))
      else WRONG "parameter type mismatch"
  | iterate2 (DEC (PR_PROP (t0, t1, bs), C))
    = iterate2 (refocus (CLO_APP (CLO_GND (t0, bs),
                                  CLO_GND (t1, bs)), C))
  | iterate2 (DEC (PR_APP (t1, t2), C))
    = WRONG "non-function application"

fun normalize2 t
    = iterate2 (refocus (CLO_GND (term_to_hterm t, TEnv.empty), CTX_MT))

fun type_check t
    = case normalize2 t
       of (RESULT v) => v
        | WRONG s => T_ERROR s
```

3.4.3 Lightweight fusion: from small-step to big-step abstract machine

The next step is to *fuse* the definitions of `iterate2` and `refocus` from the previous section. The result of the fusion, called `iterate3`, is directly applied to the result of `decompose_closure` and `decompose_context`. The result is a big-step abstract machine consisting of three mutually tail-recursive state-transition functions [59]:

- `refocus3_closure`, the composition of `iterate2` and `decompose_closure` and a clone of `decompose_closure`,
- `refocus3_context`, the composition of `iterate2` and `decompose_context`, which directly calls `iterate3` over the value of decomposition,
- `iterate3`, a clone of `iterate2` that calls the fused function `refocus3_closure`.

The resulting implementations of the functions look as follows:

```
(*  refocus3_closure : closure * hctx -> result *)
fun refocus3_closure (CLO_NUM, C)
    = refocus3_context (C, T_NUM)
  | refocus3_closure (CLO_ARR_TYPE v, C)
    = refocus3_context (C, v)
  | refocus3_closure (CLO_GND (H_LIT n, bs), C)
    = refocus3_context (C, T_NUM)
  | refocus3_closure (CLO_GND (H_IDE x, bs), C)
    = iterate3 (DEC (PR_IDE (x, bs), C))
  | refocus3_closure (CLO_GND (H_LAM (x, t, e), bs), C)
    = iterate3 (DEC (PR_LAM (x, t, e, bs), C))
  | refocus3_closure (CLO_GND (H_APP (t0, t1), bs), C)
    = iterate3 (DEC (PR_PROP (t0, t1, bs), C))
  | refocus3_closure (CLO_GND (H_TNUM, bs), C)
    = refocus3_context (C, T_NUM)
  | refocus3_closure (CLO_GND (H_TARR (t, e), bs), C)
    = refocus3_closure (CLO_GND (e, bs),
                        CTX_ARR (t, C))
  | refocus3_closure (CLO_APP (c0, c1), C)
    = refocus3_closure (c0, CTX_FUN (C, c1))
  | refocus3_closure (CLO_ARR (v, c), C)
    = refocus3_closure (c, CTX_ARR (v, C))


(*  refocus3_context : hctx * typ -> result  *)
and refocus3_context (CTX_MT, v)
    = iterate3 (VAL v)
  | refocus3_context (CTX_FUN (C, c1), v0)
    = refocus3_closure (c1, CTX_ARG (v0, C))
  | refocus3_context (CTX_ARG (v0, C), v1)
    = iterate3 (DEC (PR_APP (v0, v1), C))
  | refocus3_context (CTX_ARR (v0, C), v1)
    = iterate3 (DEC (PR_ARR (v0, v1), C))

(*  iterate3 : type_or_decomposition -> result  *)
and iterate3 (VAL v)
    = RESULT v
  | iterate3 (DEC (PR_NUM, C))
    = refocus3_closure (CLO_NUM, C)
  | iterate3 (DEC (PR_ARR (t1, t3), C))
    = refocus3_closure (type_to_closure
                (T_ARR (t1, t3)), C)
  | iterate3 (DEC (PR_IDE (x, bs), C))
    = (case TEnv.lookup (x, bs)
        of NONE
           => WRONG "undeclared identifier"
         | (SOME v) =>
           refocus3_closure (type_to_closure v, C))
  | iterate3 (DEC (PR_LAM (x, t, e, bs), C))
    = refocus3_closure (CLO_GND (H_TARR (t, e),
                  TEnv.extend (x, t, bs)), C)
  | iterate3 (DEC (PR_APP (T_ARR (t1, t3), v), C))
    = if t1 = v
```

```
      then refocus3_closure ((type_to_closure t3), C)
      else WRONG "parameter type mismatch"
  | iterate3 (DEC (PR_PROP (t0, t1, bs), C))
    = refocus3_closure (CLO_APP (CLO_GND (t0, bs),
                                 CLO_GND (t1, bs)), C)
  | iterate3 (DEC (PR_APP (t1, t2), C))
    = WRONG "non-function application"

(*  normalize3 : term -> result  *)
fun normalize3 t
    = refocus3_closure (CLO_GND (term_to_hterm t,
                                 TEnv.empty), CTX_MT)

fun type_check t
    = case normalize3 t
       of (RESULT v) => v
        | WRONG s  => T_ERROR s
```

3.4.4 Compressing corridor transitions

In the abstract machine from the previous section many transitions are *corridors*, i.e., they yield configurations for which there is a unique place for further consumption. In this section, we *compress* these configurations by inlining the corresponding consumers' clauses. We copy the functions from the previous sections, changing their indices from 3 to 4.

We are also taking advantage of the equivalence between `refocus4_closure (embed_value_in_closure v, C)` and `refocus4_context (C, v)`.

For example, the clause `refocus4_closure (CLO_GND (H_LAM (x, t, e), bs), C)` is refactored as follows:

```
refocus4_closure (CLO_GND (H_LAM (x, t, e), bs), C)
= (* by unfolding the call to refocus4_closure *)
iterate4 (DEC (PR_LAM (x, t, e, bs), C))
= (* by unfolding the call to iterate4 *)
refocus4_closure (CLO_GND (H_TARR (t, e),
        TEnv.extend (x, type_to_value t, bs)), C)
```

So the resulting implementations look as follows:

```
fun refocus4_closure (CLO_GND (H_LIT n, bs), C)
    = refocus4_context (C, T_NUM)
  | refocus4_closure (CLO_GND (H_IDE x, bs), C)
    = (case TEnv.lookup (x, bs)
        of NONE
           => WRONG "undeclared identifier"
         | (SOME v) =>
```

```
            refocus4_context (C, v))
  | refocus4_closure (CLO_GND (H_LAM (x, t, e), bs), C)
    = refocus4_closure (CLO_GND (H_TARR (t, e),
                                 TEnv.extend (x, t, bs)), C)
  | refocus4_closure (CLO_GND (H_APP (t0, t1), bs), C)
    = refocus4_closure (CLO_GND (t0, bs), CTX_FUN (C, CLO_GND (t1, bs)))
  | refocus4_closure (CLO_GND (H_TNUM, bs), C)
    = refocus4_context (C, T_NUM)
  | refocus4_closure (CLO_GND (H_TARR (t, e), bs), C)
    = refocus4_closure (CLO_GND (e, bs), CTX_ARR (t, C))

and refocus4_context (CTX_MT, v)
    = RESULT v
  | refocus4_context (CTX_FUN (C, c1), v0)
    = refocus4_closure (c1, CTX_ARG (v0, C))
  | refocus4_context (CTX_ARG (v0, C), v1)
    = iterate4 (v0, v1, C)
  | refocus4_context (CTX_ARR (v0, C), v1)
    = refocus4_context (C, (T_ARR (v0, v1)))

and iterate4 (T_ARR (t1, t4), v, C)
    = if t1 = v
      then refocus4_context (C, t4)
      else WRONG "parameter type mismatch"
  | iterate4 (t, v, C)
    = WRONG "non-function application"

fun normalize1 t
    = refocus4_closure (CLO_GND (term_to_hterm t, TEnv.empty), CTX_MT)

fun type_check t
    = case normalize1 t
       of (RESULT v) => v
        | WRONG s  => T_ERROR s
```

After this transformation *all* clauses of the function `refocus4_closure` for non-ground closures are now dead, i.e., unused, and therefore can be safely removed. One can also notice that all transitions of `refocus4_closure` are now over ground closures, so we can flatten them by peeling off the closure part.

3.4.5 Renaming transition functions and flattening configurations

The resulting simplified machine is a familiar 'eval/apply/continue' abstract machine operating over ground closures. For this section we rename `refocus4_closure` to `eval5`, `refocus4_context` to `continue5` and `iterate4` to `apply5`. We flatten the configuration of `refocus4_closure` as well as definitions of values and contexts.

Therefore, closures are no longer involved in computations, and the former hybrid contexts and abstract machine transition functions look now as follows:

```
datatype context = CTX_MT
                 | CTX_FUN of context * hterm * bindings
                 | CTX_ARG of typ * context
                 | CTX_ARR of typ * context

datatype result = RESULT of typ
                | WRONG of string

(*  eval5 : hterm * (string * typ) list * context -> result  *)
fun eval5 (H_LIT n, gamma, C)
    = continue5 (C, T_NUM)
  | eval5 (H_IDE x, gamma, C)
    = (case TEnv.lookup (x, gamma)
        of NONE
           => WRONG "undeclared identifier"
         | (SOME v) =>
           continue5 (C, v))
  | eval5 (H_LAM (x, t, e), gamma, C)
    = eval5 (H_TARR (t, e), TEnv.extend (x, t, gamma), C)
  | eval5 (H_APP (t0, t1), gamma, C)
    = eval5 (t0, gamma, CTX_FUN (C, t1, gamma))
  | eval5 (H_TNUM, gamma, C)
    = continue5 (C, T_NUM)
  | eval5 (H_TARR (t, e), gamma, C)
    = eval5 (e, gamma, CTX_ARR (t, C))

(*  continue5 : context * typ -> result  *)
and continue5 (CTX_MT, v)
    = RESULT v
  | continue5 (CTX_FUN (C, c1, gamma), v0)
    = eval5 (c1, gamma, CTX_ARG (v0, C))
  | continue5 (CTX_ARG (v0, C), v1)
    = apply5 (v0, v1, C)
  | continue5 (CTX_ARR (v0, C), v1)
    = continue5 (C, (T_ARR (v0, v1)))

(*  apply5 : typ * typ * context -> result  *)
and apply5 (T_ARR (t1, t4), v, C)
    = if t1 = v
      then continue5 (C, t4)
      else WRONG "parameter type mismatch"
  | apply5 (t, v, C)
    = WRONG "non-function application"

(*  normalize5 : term -> result  *)
fun normalize5 t = eval5 (term_to_hterm t, TEnv.empty, CTX_MT)

(*  type_check : term -> typ  *)
fun type_check t
    = case normalize5 t
```

```
       of (RESULT v) => v
        | WRONG s => T_ERROR s
```

3.4.6 Removing hybrid artifacts and switching domains

The next simplification is to remove $\lambda_{\mathcal{H}}$-related artifacts from machine configurations. We copy functions from the previous section and perform some extra corridor transition compressions. For instance,

```
eval5 (H_LAM (x, t, e), gamma, C)
= (* by unfolding the definition of eval5 *)
eval5 (H_TARR (t, e), TEnv.extend (x, type_to_value t, gamma), C)
= (* by unfolding the definition of eval5 *)
eval5 (e, TEnv.extend (x, type_to_value t, gamma),
       CTX_ARR (type_to_value t, C))
```

As a result, there are no more clauses mentioning elements of the hybrid language such as `H_TNUM` (removed as an unused clause of `eval5`) and `H_TARR`. So now we can switch the domain of the `eval5`, `continue5` and `apply5` functions from `hterm` to `term`. The second observation is that algebraic data type `result` is in fact isomorphic to the data type `typ`, so we can switch the domain of values as well as follows:

$$\begin{array}{rcl} \mathsf{RESULT}\ (\mathsf{T\_NUM}) & \mapsto & \mathsf{T\_NUM} \\ \mathsf{RESULT}\ (\mathsf{T\_ARR}\ (\tau_1,\tau_2)) & \mapsto & \mathsf{T\_ARR}\ (\tau_1,\tau_2) \\ \mathsf{WRONG}\ (s) & \mapsto & \mathsf{T\_ERROR}\ (s) \end{array}$$

The domain switching changes the signature of the function `eval` and yields the following implementation.

```
datatype result = RESULT of typ
                | WRONG of string

datatype context = CTX_MT
                 | CTX_FUN of context * term * bindings
                 | CTX_ARG of typ * context
                 | CTX_ARR of typ * context

(*  term * (string * typ) list * context -> typ  *)
fun eval6 (LIT n, gamma, C)
    = continue6 (C, T_NUM)
  | eval6 (IDE x, gamma, C)
    = (case TEnv.lookup (x, gamma)
        of NONE
           => T_ERROR "undeclared identifier"
         | (SOME v) =>
```

```
            continue6 (C, v))
  | eval6 (LAM (x, t, e), gamma, C)
    = eval6 (e, TEnv.extend (x, t, gamma),
               CTX_ARR (t, C))
  | eval6 (APP (t0, t1), gamma, C)
    = eval6 (t0, gamma, CTX_FUN (C, t1, gamma))

(*  continue6 : context * typ -> typ  *)
and continue6 (CTX_MT, v)
    = v
  | continue6 (CTX_FUN (C, c1, gamma), v0)
    = eval6 (c1, gamma, CTX_ARG (v0, C))
  | continue6 (CTX_ARG (v0, C), v1)
    = apply6 (v0, v1, C)
  | continue6 (CTX_ARR (v0, C), v1)
    = continue6 (C, (T_ARR (v0, v1)))

(*  apply6 : typ * typ * context -> typ  *)
and apply6 (T_ARR (t1, t4), v, C)
    = if t1 = v
      then continue6 (C, t4)
      else T_ERROR "parameter type mismatch"
  | apply6 (t, v, C)
    = T_ERROR "non-function application"

(*  term -> typ  *)
fun normalize6 t
    = eval6 (t, TEnv.empty, CTX_MT)

fun type_check t
    = normalize6 t
```

This might come as a surprise, but the resulting abstract machine is the well-known environment-based CEK machine [80]. `eval6`'s argument of type `term * (string * typ) list * context` corresponds to a control state, where the control string is represented by terms fo type `term`, environments are of type `(string * typ) list` and the continuation component K is represented by the type `context`.

3.4.7 Refunctionalization

The abstract machine obtained in the previous section is in fact in defunctionalized form [62]: the reduction contexts, together with `continue6`, are the first-order counterpart of continuations. To obtain the higher-order counterpart we use a technique known as *refunctionalization* [60]. The resulting refunctionalized program is a compositional evaluation function in continuation-passing style.

```
(*  eval7 : term * (string * typ) list * (typ -> typ) -> typ  *)
fun eval7 (LIT n, gamma, k)
    = k T_NUM
  | eval7 (IDE x, gamma, k)
    = (case TEnv.lookup (x, gamma)
        of NONE
           => T_ERROR "undeclared identifier"
         | (SOME v) =>
           k v)
  | eval7 (LAM (x, t, e), gamma, k)
    = eval7 (e, TEnv.extend (x, t, gamma),
          fn v => k (T_ARR (t, v)))

  | eval7 (APP (e0, e1), gamma, k)
    = eval7 (e0, gamma,
          fn t => case t
                    of T_ARR (t1, t2)
                       => eval7 (e1, gamma,
                              fn v1 =>
                                 if t1 = v1
                                 then k t2
                                 else T_ERROR "parameter type mismatch")
                     | _ => T_ERROR "non-function application")

(*  normalize7 : term -> typ  *)
fun normalize7 t
    = eval7 (t, TEnv.empty, fn x => x)

fun type_check t
    = normalize7 t
```

3.4.8 Back to direct style

The refunctionalized definition from the previous section is in continuation-passing style: it has a functional accumulator and all of its calls are tail calls. To implement it in direct style in the presence of non-local returns in cases where typing error occurs, the library for undelimited continuations `SMLofNJ.Cont`, provided by Standard ML of New Jersey, is used.

```
val callcc = SMLofNJ.Cont.callcc
val throw = SMLofNJ.Cont.throw

(*  normalize8: term -> typ  *)
fun normalize8 t = callcc (fn top =>
  let fun eval8 (LIT n, gamma) = T_NUM
    | eval8 (IDE x, gamma) = (case TEnv.lookup (x, gamma)
            of NONE     => throw top (T_ERROR "undeclared identifier")
             | (SOME v) => v)
    | eval8 (LAM (x, t, e), gamma)
```

```
      = T_ARR (t, eval8 (e, TEnv.extend (x, t, gamma)))
    | eval8 (APP (e0, e1), gamma)
      = let val t = eval8 (e0, gamma)
            val v1 = eval8 (e1, gamma)
        in (case t of T_ARR (t1, t2)
                 => if t1 = v1 then t2
                    else throw top (T_ERROR "parameter type mismatch")
             | _ => throw top (T_ERROR "non-function application"))
        end
  in eval8 (t, TEnv.empty)
  end)

(*  type_check: term -> typ  *)
fun type_check t = normalize8 t
```

The resulting program is a traditional evaluator for type checking, such as the one described by Pierce [164, pages 113-116]. This implementation also corresponds straightforwardly to the type checking rules à la Church, as defined in Figure 3.1.

The only one difference is that our implementation uses undelimited continuations via `callcc` to propagate encountered type errors whereas a classical implementation would just perform some additional check in each clause of the `eval` function or use the exceptions. This last transition completes the chain of transformations and the chapter.

Chapter 4

From Type Checking via Evaluation to Type Checking with an Abstract Machine

In this chapter, we continue the story about the mechanical inter-derivation of type checking semantics and demonstrate an application of techniques investigated by Danvy et al. to derive an *abstract machine* for typing from the traditional recursive descent approach. Again, all techniques we are going to use are off-the-shelf and *no* appropriate correspondence theorems between an initial type system and the derived abstract machine needs to be proven as they follow directly from the correctness of inter-derivation and of the initial specification.

A recursive descent-based implementation, which we obtained at the end of Chapter 3, is something straightforward to implement based on declarative typing rules (Figure 1.1, see Chapter 1). The abstract machine we will derive in this chapter exposes behaviour similar to Landin's SECD machine [124] and gives a solid basis for further optimizations using abstract interpretation.

4.1 Type-Checking Abstract Machines

The first step towards connecting type inference by recursive descent with type inference via abstract machines is due to Hankin and Le Métayer [98]. The authors provided a description of an abstract machine-like formalism for implementing type

checking and type inference systems. The described technique is in the spirit of Hannan and Miller [99] and yields in several stages an abstract machine based on the given type derivation rules. That machine strongly resembles Landin's SECD machine [124]. The only difference between the resulting machine and the original SECD machine is that the former has no "D" component since there is no "dump" in the corresponding evaluator, so we call the respective artifact *SEC machine* following the tradition to name machines after their control strings. A simplified version of the small-step machine, defined by its state space and the transition relation, is given in Figure 4.1.

We use the standard notation for stack, where nil denotes an empty stack and $c :: C$ is equivalent to the operation *push* taking some element c and a list C. By abusing the notation, we also use the construct $c :: C$ to perform pattern-matching on non-empty stacks with a top element c. We use an underscore "_" to denote any possible element in the left-hand side of the transition relation. The typing environment *lookup* $E[x \mapsto \tau]$ matches an environment E, such that $E = E', x : \tau, E''$ for some environments E' and E'', such that E'' does not bind x. Finally, an environment *extension* $E \sqcup \{x \mapsto \tau\}$ returns a new environment $E', x : \tau$. Names of transition rules are consistent with the prototype implementation from Section A.2 of Appendix A, which we used to generate Figure 1.3.

We use the standard notation $\Rightarrow_t^*$ to express the reflexive-transitive closure of the relation $\Rightarrow_t$. The following theorem has been proven for soundness and completeness of the derived machine:

Theorem 4.1.1. [98] (Soundness and Completeness for $\Rightarrow_t$)
$\Gamma \vdash e : \tau$ *iff* $\langle S, \Gamma, e :: C \rangle \Rightarrow_t^* \langle \tau : S, \Gamma, C \rangle$.

One can see that the third component of the abstract machine (i.e., "C" for *control*) contains λ-terms as control elements, but also specific tokens, such as *Lam*, *Fun* and *Arg*, with extra bits of context information. Intuitively it is clear that these elements correspond somehow to combination of type constituents in derivation rules. However, the question that was open until now is what is a formal meaning of this correspondence?

In this work, we describe a staged *mechanical* inter-derivation of the two above mentioned type inference procedures via the program transformations used in Reynolds's functional correspondence between evaluators and big-step abstract machines [2, 174] and in Danvy et al.'s work on the systematic deconstruction of Landin's SECD machine [58]. The correspondence between a traditional type system and a SEC machine for type inference is provided by the construction and inter-derivation of their computational counterparts. The pleasant consequence is that no soundness and completeness theorems need to be proven: they are instead corollaries of the correctness of inter-derivation and of the initial specification [49].

Expressions	e	$::=$	$n \mid x \mid \lambda x : \tau.e \mid e\ e$
Types	τ	$::=$	$\mathsf{num} \mid \tau \to \tau$
Results stack	S	$::=$	$\mathsf{nil} \mid \tau :: S$
Control elements	c	$::=$	$e \mid \mathit{Lam}(\tau,S) \mid \mathit{Fun}(e) \mid \mathit{Arg}(\tau,\tau)$
Control stacks	C	$::=$	$\mathsf{nil} \mid c :: C$
Typing environments	E	$::=$	$\emptyset \mid E, x : \tau$

Abstract machine state-space

$$
\begin{array}{rcll}
\langle S, E, n :: C\rangle & \Rightarrow_t & \langle \mathsf{num} :: S, E, C\rangle & [\mathsf{num}] \\
\langle S, E[x \mapsto \tau], x :: C\rangle & \Rightarrow_t & \langle \tau :: S, E[x \mapsto \tau], C\rangle & [\mathsf{var}] \\
\langle S, E, (\lambda x : \tau.e) :: C\rangle & \Rightarrow_t & \langle \mathsf{nil}, E \sqcup \{x \mapsto \tau\}, e : \mathit{Lam}(\tau, S) :: C\rangle & [\mathsf{lam}] \\
\langle S, E, (e_1\ e_2) :: C\rangle & \Rightarrow_t & \langle S, E, e_1 :: \mathit{Fun}(e_2) :: C\rangle & [\mathsf{app}] \\
\langle \tau_2 :: S, E, \mathit{Lam}(\tau_1, S') :: C\rangle & \Rightarrow_t & \langle (\tau_1 \to \tau_2) :: S', E, C\rangle & [\mathsf{t\text{-}lam}] \\
\langle (\tau_1 \to \tau_2) :: S, E, \mathit{Fun}(e_2) :: C\rangle & \Rightarrow_t & \langle (\tau_1 \to \tau_2) :: S, E, e_2 : \mathit{Arg}(\tau_1, \tau_2) :: C\rangle & [\mathsf{t\text{-}fun}] \\
\langle \tau_1 :: \_ :: S, E, \mathit{Arg}(\tau_1, \tau_2) :: C\rangle & \Rightarrow_t & \langle \tau_2 :: S, E, C\rangle & [\mathsf{t\text{-}arg}]
\end{array}
$$

Transition rules

Figure 4.1: A small-step abstract machine for type checking à la Hankin and Le Métayer [98].

4.1.1 Chapter outline

The remainder of this chapter is structured as follows. Section 4.2 gives an overview of our method, enumerating the techniques involved. Section 4.3 provides the implementation of type checking of the simply typed lambda calculus and describes the initial setting for further functional transformations. Section 4.4 describes the set of program transformations corresponding to the construction of an abstract machine for type inference from the traditional type inference procedure in the form of a recursive descent.

4.2 Method Overview

A diagram with an overview of program transformations is shown in Figure 4.2.

We provide an implementation of a traditional type checker for the STLC in the form of a recursive descent as a starting point for further transformations (Section 4.3).

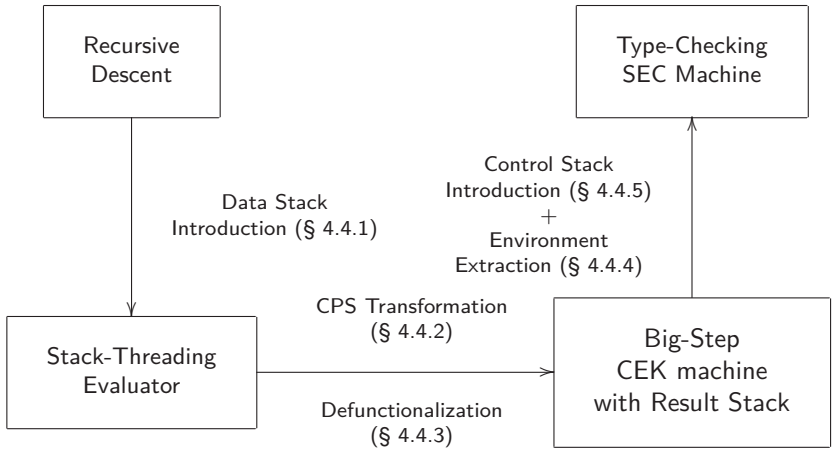

Figure 4.2: Inter-derivation from a compositional type checker to a type-checking SEC machine

We successively refactor it into a stack-threading *callee-save* evaluator, i.e., one that pushes its results on an explicit local stack, which is passed around as a parameter — the component "S" of a control string (Section 4.4.1). The obtained evaluator is in non-tail call form, so we transform it into continuation-passing style (Section 4.4.2) and then defunctionalize it (Section 4.4.3), which leads to the big-step stack-threading CEK machine. The type environment is still a part of some defunctionalized contexts, so we extract it as an explicit parameter of the evaluator, i.e., the component "E" of the control string (Section 4.4.4). We introduce an explicit control stack (the component "C" of the control string) in order to merge together several mutually recursive transition functions (Section 4.4.5), which yields a big-step SEC machine. Finally, we rework the big-step machine into a small-step one by extracting an *iteration* function (Section 4.4.6). The final machine is Landin's SECD machine lacking the "D" component of its control string, since no explicit control flow management with *dumps* is needed for type-checking.

4.3 Initial Setting: Type Checking via Recursive Descent

This section provides the initial implementation of a type checking procedure for the simply typed lambda calculus, which will be used for further transformations in Section 4.4.

4.3.1 Terms and types

The abstract syntax of simply typed lambda calculus includes integer literals, identifiers, lambda-abstractions and applications. Types are either numeric types or arrow types. The special value `T_ERROR s` is used for typing errors and cannot be a constituent of any other type. We implement terms and types with the following SML data types:

```
datatype term = LIT of int
              | IDE of string
              | LAM of string * typ * term
              | APP of term * term

datatype typ = T_NUM
             | T_ARR of typ * typ
             | T_ERROR of string
```

4.3.2 Type checking procedure

Typing environments `TEnv` represent bindings of identifiers to types. They carry typing assumptions about free variables in λ-terms. The value `empty` corresponds to an empty environment, `extend` extends an environment with a new binding of a variable into a type and, the function `lookup` extracts the typing assumption, associated with a particular variable. A lookup may fail, which is reflected by its return type `'a option`.[1]

```
signature TEnv =
sig
  type 'a gamma
  val empty : (string * 'a) gamma
  val extend : string * 'a * (string * 'a) gamma ->
               (string * 'a) gamma
  val lookup : string * (string * 'a) gamma -> 'a option
end
```

The canonical procedure for type checking [164, pages 113-116] is implemented as a recursive descent as follows.

```
exception TYPING_ERROR of string

(* check0 : term * typ gamma -> typ *)
fun check0 (LIT n, e)
```

[1] In order to keep to a uniform approach for different semantics for type inference (see [123, 183] or Chapter 3 of this dissertation), we leave environments parametrized by the type parameter `'a`, which is instantiated with `typ` in this case.

```
    = T_NUM
  | check0 (IDE x, e)
    = case TEnv.lookup(x, e)
        of (SOME t) => t
         | NONE     => raise (TYPING_ERROR "undeclared identifier")
  | check0 (LAM (x, arg_type, body), e)
    = let val body_type = check0 (body,
          (TEnv.extend (x, arg_type, e)))
      in T_ARR (arg_type, body_type)
      end
  | check0 (APP (e1, e2), e)
    = case check0 (e1, e)
        of T_ARR (t1, t2) =>
          let val arg_type = check0 (e2, e)
          in  if arg_type = t1
              then t2
              else raise (TYPING_ERROR "parameter type mismatch")
          end
        |  _ => raise (TYPING_ERROR "non-function application")

(*  type_check : term -> typ  *)
fun type_check t = check0 (t, TEnv.empty)
```

4.3.3 Representation of typing errors

Three kinds of typing errors might occur during type checking:

- ***Undefined identifier in an environment***, corresponds to the `MatchError` exception of SML thrown in the second clause of `check0` function.
- ***Non-arrow type in a function position***, is represented by a `MatchError` raised at the top-level of `check0` function
- ***A type mismatch between a function parameter type and an argument type***, a `TYPING_ERROR` exception is thrown at the last clause of `check0` function.

Moreover, there are at least three different ways to represent typing errors in practice and propagate the information about them to the client of the type checker.

1. *"Bubbling up"*: for results of recursive calls, the type-checking procedure checks explicitly whether the result is a type error. If it is, this information is returned immediately to an "upper level".
2. *Using exceptions*: when a type error occurs, the necessary information for the client of the type checker can be put into an exception, which is immediately

thrown. This approach is employed in the described implementation of the function `check0`.

3. *Continuation dropping*: if the type-checker is in continuation-passing style, one can interrupt the current control flow in case of a typing error. Then an error value will be returned instead of applying the continuation to the result [53].

In the following series of transformations we will be switching between the second and the third approaches.

4.4 From Recursive Descent to SEC Machine

The derivational approach we describe in this section takes advantage of Reynolds's functional correspondence between different ways to represent semantic artifacts [174] and more recent work by Danvy et al. on the deconstruction of Landin's SECD machine [58].

4.4.1 Extracting a result stack

In the canonical implementation of a type checker the results of nested calls of the `check0` function are allocated on local stack frames of callees. We represent this model explicitly by introducing local result stacks and passing them around as an explicit parameter of the `check1` function. A data stack, which is the "S" component of a control string for the machine presented in Section 4.1, stores intermediate values after they have been computed but before they are used. Computing an expression leaves its value on top of the data stack. Applications expect to find their argument and a function on top of this data stack. In case of nested calls the immutable part of the stack is saved by the callee, and the caller is invoked with a reduced or fresh stack. This kind of evaluator is classified as a *callee-save*,[2] explicit stack-threading one according to Danvy and Millikin [58, Appendix D].

The implementation of the function `type_check` is changed correspondingly to take the head of the result list as the result of a computation.

```
(* check1 : term * typ list * typ gamma -> typ list *)
fun check1 (LIT n, s, e)
    = T_NUM :: s
  | check1 (IDE x, s, e)
    = case TEnv.lookup(x, e)
```

[2] The terminology is due to the convention according to which the local environment and the result stack are saved and restored by the *callee* subroutine.

```
        of (SOME t) => t :: s
         | NONE     => raise (TYPING_ERROR "undeclared identifier")
  | check1 (LAM (x, arg_type, body), s, e)
    = let val (body_type :: _) =
              check1 (body, nil, (TEnv.extend (x, arg_type, e)))
      in T_ARR (arg_type, body_type) :: s
      end
  | check1 (APP (e1, e2), s, e)
    = case check1 (e1, nil, e)
        of s0 as (T_ARR (t1, t2) :: _) =>
           let val s0 as (T_ARR (t1, t2) :: _) = check1 (e1, nil, e)
               val (arg_type :: x :: _) = check1 (e2, s0, e)
           in if arg_type = t1
              then t2 :: s
              else raise (TYPING_ERROR "parameter type mismatch")
           end
         | _ => raise (TYPING_ERROR "non-function application")

(*  type_check : term -> typ  *)
fun type_check t
  = let val (v :: s) =  check1 (t, nil, TEnv.empty)
    in v end
```

4.4.2 CPS transformation

The function `check1` from the previous section is transformed into continuation-passing style (CPS). This is done in three steps, as described in Danvy's report [46]. Briefly, each intermediate result of a computation is extracted into a new local variable, their computations are sequentialized and a new formal parameter, namely, a continuation is introduced. Thus the intermediate results are named by the formal parameters of each of the lambda-abstractions that define the continuation.

```
(*  check2 : term * typ list * typ gamma *
             (typ list -> typ list) -> typ list  *)
fun check2 (LIT n, s, e, k)
    = k (T_NUM :: s)
  | check2 (IDE x, s, e, k)
    = k (case TEnv.lookup(x, e)
           of (SOME t) => t :: s
            | NONE     => (T_ERROR "undeclared identifier") :: nil)
  | check2 (LAM (x, arg_type, body), s, e, k)
    = check2 (body, nil, (TEnv.extend (x, arg_type, e)),
           fn (body_type :: s0) =>
              k (T_ARR (arg_type, body_type) :: s))
  | check2 (APP (e1, e2), s, e, k)
    = check2 (e1, nil, e,
        fn s0 =>
           case s0
             of (T_ARR (t1, t2) :: _) =>
```

```
                  check2 (e2, s0, e,
                    fn (arg_type :: x :: _) =>
                       if arg_type = t1
                       then k (t2 :: s)
                       else (T_ERROR "parameter type mismatch") :: nil)
                | _ => (T_ERROR "non-function application") :: nil)

(*  type_check : term -> typ  *)
fun type_check t
  = let val (v :: s) =  check2 (t, nil,
                                TEnv.empty, fn x => x)
    in v end
```

Since we have CPS-transformed our program, we may replace exception raising by non-local returns, as it is done now in the last clause of `check2` function: a `T_ERROR` is returned directly if a typing error occurs. This small transformation corresponds to the switching between the second and third methods of typing error representation described in Section 4.3. The resulting procedure, considered as an interpreter of λ-terms, is a traditional continuation-passing one.

4.4.3 Defunctionalization

The next step is to defunctionalize the continuations in the implementation of the type checker from Section 4.4.2. The function space of the program under consideration is inhabited by the four function values that arise from considering four function abstractions from the definitions of functions `check2` and `type_check`: one initial continuation in `type_check` and three more in two last clauses of `check2`. We therefore partition the function space into four summands and represent it as the following first-order data type:

```
datatype cont = CONT_MT
              | CONT_LAM of typ * cont *  typ list
              | CONT_FUN of cont * term * typ gamma
              | CONT_ARG of typ * typ * cont
```

Those defunctionalized continuations represent first-order *evaluation contexts* of type computations on top of the abstract syntax of the calculus. Contexts are produced at places of former lambda-abstractions (the initial call of the function `type_check` and third and forth clauses of the function `check2`) and consumed by a dispatcher-like function `continue3`.

```
(*  check3 : term * typ list * typ gamma * cont ->
             typ list *)
fun check3 (LIT n, s, e, C)
    = continue3 (C, (T_NUM :: s))
```

```
  | check3 (IDE x, s, e, C)
    = continue3 (C,
        (case TEnv.lookup(x, e)
           of (SOME t) => t :: s
            | NONE     => (T_ERROR "undeclared identifier") :: nil))
  | check3 (LAM (x, arg_type, body), s, e, C)
    = check3 (body, nil, (TEnv.extend (x, arg_type, e)),
              CONT_LAM (arg_type, C, s))
  | check3 (APP (e1, e2), s, e, C)
    = check3 (e1, s, e, CONT_FUN (C, e2, e))

(*  continue3 : cont * typ list -> typ list  *)
and continue3 (CONT_MT, s)
    = s
  | continue3 (CONT_LAM (arg_type, C, s), (body_type :: s0))
    = continue3 (C, T_ARR (arg_type, body_type) :: s)
  | continue3 (CONT_FUN (C, e2, e), s0 as (T_ARR (t1, t2) :: _))
    = check3 (e2, s0, e, CONT_ARG (t1, t2, C))
  | continue3 (CONT_FUN (C, e2, e), _)
    = (T_ERROR "non-function application") :: nil
  | continue3 (CONT_ARG (t1, t2, C), (arg_type :: x :: s1))
    = if arg_type = t1
      then continue3 (C, t2 :: s1)
      else (T_ERROR "parameter type mismatch") :: nil

(*  type_check : term -> typ  *)
fun type_check t
  = let val (v :: s) = check3 (t, nil,
                               TEnv.empty, CONT_MT)
    in v end
```

The resulting machine is an analogue of the well-known environment-based CEK machine with an explicit component `s` for the result stack [80]. Each tail call implements a state transition of the machine.

4.4.4 Extracting the environment as a parameter

One can notice that the type environment is part of the data type of evaluation contexts. We massage the type checking machine by extracting an environment to a separate explicit parameter of the function `continue4`. It will correspond to the component E in the control string of the final abstract machine. Now the constructor `CONT_FUN`, which is consumed by `continue4`, does not contain an environment as a parameter. We also rearrange parameters of the data type `cont` to give them a list shape. The data type of contexts is now as follows:

```
datatype cont = CONT_MT
              | CONT_LAM of typ * typ list * cont
              | CONT_FUN of term * cont
              | CONT_ARG of typ * typ * cont
```

The next natural step is to take advantage of the list-like structure of contexts represented by `cont`.

4.4.5 Adding an explicit control stack

In this section we introduce the last component of the control string of the abstract machine, namely, the *control stack* C. The defunctionalized contexts from the Section 4.4.4 expose a stack-like structure with `CONT_MT` as the empty element. The structure can be refactored into a stack of control tokens, corresponding to the particular summands of `cont`. To unify the structure of states we also introduce one more extra control token for terms. Control stack tokens are represented by the following data structure:

```
datatype control_element = C_ARG of typ * typ
                         | C_FUN of term
                         | C_LAM of typ * typ list
                         | C_TERM of term
```

Former `CONT_MT` element corresponds now to an empty control stack. Since the domain of control elements is now "lifted" to `control_element`, we may safely merge the functions `continue4` and `check4` to get the unified function `check5`.

```
(*  check5 : typ list * typ gamma * control_element list
             -> typ list  *)
fun check5 (s, e, C_TERM (LIT n) :: C)
    = check5 (T_NUM :: s, e, C)
  | check5 (s, e, C_TERM (IDE x) :: C)
    = check5 (case TEnv.lookup(x, e)
               of (SOME t) => t :: s
                | _        => (T_ERROR "undeclared identifier") :: nil,
              e, C)
  | check5 (s, e, C_TERM (LAM (x, arg_type, body)) :: C)
    = check5 (nil, TEnv.extend (x, arg_type, e),
              C_TERM body :: C_LAM (arg_type, s) :: C)
  | check5 (s, e, C_TERM (APP (e1, e2)) :: C)
    = check5 (s, e, C_TERM e1 :: C_FUN e2 :: C)
  | check5 ((body_type :: s0), e, C_LAM (arg_type, s) :: C)
    = check5 (T_ARR (arg_type, body_type) :: s, e, C)
  | check5 (s0 as (T_ARR (t1, t2) :: _), e, C_FUN e2 :: C)
    = check5 (s0, e, C_TERM e2 :: C_ARG (t1, t2) :: C)
  | check5 (_, e, C_FUN e2 :: C)
    = (T_ERROR "non-function application") :: nil
```

```
  | check5 (v2 :: x :: s1, e,
            C_ARG (arg_type, result_type) :: C)
    = if v2 = arg_type
      then check5 (result_type :: s1, e, C)
      else T_ERROR "parameter type mismatch" :: nil
  | check5 (s, e, nil)
    = s

(*  type_check : term -> typ  *)
and type_check t
  = let val (v :: s) =  check5 (nil, TEnv.empty,
                                C_TERM t :: nil)
    in v end
```

The resulting interpreter is a big-step SEC machine where each tail call of `check5` corresponds to a transition. Now we are going to turn it into a small-step machine by introducing an explicit driver-loop function.

4.4.6 From a big-step to a small-step SEC machine

Since the big-step SEC machine from Section 4.4.5 has only one type of control string, it is straightforward to transform it into a small-step machine by introducing a dedicated driver-loop function `iterate6`:

```
type state = typ list * typ gamma * control_element list

(*  step6 : state -> state  *)
fun step6 (s, e, C_TERM (LIT n) :: C)
    = (T_NUM :: s, e, C)
  | step6 (s, e, C_TERM (IDE x) :: C)
    = (case TEnv.lookup(x, e)
         of (SOME t) => (t :: s, e, C)
          | _        => raise (TYPING_ERROR "undeclared identifier"))
  | step6 (s, e, C_TERM (LAM (x, arg_type, body)) :: C)
    = (nil, TEnv.extend (x, arg_type, e),
       C_TERM body :: C_LAM (arg_type, s) :: C)
  | step6 (s, e, C_TERM (APP (e1, e2)) :: C)
    = (s, e, C_TERM e1 :: C_FUN e2 :: C)
  | step6 ((body_type :: s0), e, C_LAM (arg_type, s) :: C)
    = (T_ARR (arg_type, body_type) :: s, e, C)
  | step6 (s0 as (T_ARR (t1, t2) :: _), e, C_FUN e2 :: C)
    = (s0, e, C_TERM e2 :: C_ARG (t1, t2) :: C)
  | step6 (_, e, C_FUN e2 :: C)
    = raise (TYPING_ERROR "non-function application")
  | step6 (v2 :: x :: s1, e, C_ARG (arg_type, result_type) :: C)
    = if v2 = arg_type
      then (result_type :: s1, e, C)
      else raise (TYPING_ERROR "parameter type mismatch")
```

```
(*  iterate6 : state -> typ  *)
fun iterate6 (v :: s, _, nil)
    = v
  | iterate6 state
    = iterate6 (step6 state)

(*  type_check : term -> typ  *)
fun type_check term
  = iterate6  (nil, TEnv.empty, C_TERM term :: nil)
```

At each step of the execution the machine performs a transition to a new state and the function `iterate6` checks the termination condition. This last transition completes our chain of transformations. The transition function of the described small-step SEC machine corresponds directly to the set of transition rules given in Figure 4.1. Arrow types are consumed implicitly in the last transition rule by being popped from the result stack S with no additional check. However, the necessary check of type correspondence is performed thanks to the control element $Arg(\tau_1, \tau_2)$.

Chapter 5

Related Work and Applications

The derivations we presented in Chapters 3 and 4 are particular cases of employing functional correspondences to inter-derive semantic artifacts. Similar techniques have been widely applied during the last decade [2, 17, 49, 62, 136], and we invite the interested reader to take a look on the works [52, 51] for an exhaustive overview of applications of the technique, including, but not limited to:

- evaluators with computational effects [4];
- inter-deriving semantic artifact for object-oriented programming in the setting of Abadi and Cardelli's untyped calculus of objects [1, 56]
- systematic decomposition and rethinking of Landin's SECD machine [58];
- describing the CPS hierarchy and locating delimited continuations in it [16];
- deriving an abstract machine for the call-by-need lambda calculus [3];
- formalizing and analyzing the semantics of the Scheme programming language [18] and
- bootstrapping the method to systematically construct small-step abstract semantics for families of abstract interpretation-based static analyses [139, 204].

In this chapter, we provide a short survey of most recent and relevant work on semantic derivation focusing mostly on reduction semantics and abstract machines (Section 5.1).

We also initiate a discussion on application of the derived artifacts and the technique in general as applied to type checking algorithms (Section 5.2).

5.1 Related work

The functional correspondence between different semantics artifacts has been recently applied to various tasks. Ager et al. [4] investigate the correspondence between semantics described in terms of monadic evaluators and languages with computational effects. They show that a calculus for tail-recursive stack inspection corresponds to a lifted state monad. This correspondence allows one to combine the state monad with other monads and obtain abstract machines with both tail-recursive stack inspection and other computational effects. A similar technique applied to the standard call-by-need reduction for the λ-calculus yields a reduction-free stateless abstract machine and a heapless natural semantics for call-by-need evaluation [61]. Danvy and Zerny [66] present a purely syntactic theory of graph reduction for the canonical combinators S, K, and I, where graph vertices are represented with evaluation contexts and let expressions. This syntactic theory is expressed as a reduction semantics. Through the series of functional transformations, the authors derive a store-based abstract machine whose architecture coincides with that of Turner's original reduction machine.

More recently, Anton and Thiemann [11] took reduction semantics for different implementations of coroutines from the literature and obtained equivalent definitional interpreters by applying the same sequence of transformations we used. The operational semantics obtained is transformed further into a denotational implementation that provides the necessary basis to construct a sound type system.

Reduction semantics for type inference Reduction semantics for type checking, proposed initially by Kuan et al. [123], is another operational view on type inference algorithms. Although in this work we demonstrated the derivation only for type checking of the simply-typed lambda calculus in Church style, the same inter-derivation can be also provided with a few adjustments for the type inference algorithm à la Curry/Hindley. In fact, the reduction semantics for type inference proposed by Kuan et al. implies implicit inlining of the traditional unification function for the collected constraints, which results in a few additional reduction rules. Those rules involve the special term of the form $(\textbf{unify}\ \tau_1\ \tau_2\ p)$ for some types τ_1, τ_2 and a hybrid expression p. The types also should be extended for type variables ξ, which are a subject of unification. We believe that reverting the transformations from Chapter 3 (Figure 4.2) for the Curry/Hindley inference rules and expanding the unification traversal would give the reduction rules similar to those by Kuan et al.

5.2 Applications

The approach described in our work allows one to derive mechanically abstract machine operational representation for type derivations from their computational counterparts, implemented in the form of recursive descent. However all presented artifacts are handcrafted, the implementation of the *automatic* transformation is to be addressed in the future work. All transitions presented, except for environment extraction as we described it in Section 4.4.4, are well-known for implementors of interpreters for functional programming languages. In general, the presented technique scales to implementation of many static analyses defined compositionally in the form of derivation rules: all one needs to do is to provide a straightforward initial implementation of the appropriate recursive descent. This section discusses applications of the correspondence presented.

Type debugging Reduction semantics for type inference provides a powerful framework to implement type debuggers and improve the quality of error messages. Currently, the majority of techniques used for this task rely on program slicing [172, 201]. An explicit notion of evaluation context for type inference can provide better information for type reification based on the expected type of an expression, as it is done, for instance, in the Scala programming language [156].

Abstract machines and abstract interpretation with types The transition system described in Figure 4.1 (see Chapter 4) exhibits some generic elements, which can be adjusted according to the specific procedure of computations involving types. As it has been shown through Sections 4.4.2-4.4.5, the control stack elements *Lam*, *Fun* and *Arg* are derived from defunctionalized continuations. They trigger system-specific computations involving combinations of previously obtained types, stored in the result stack *S*.

Hankin and Le Métayer [98] in their work on lazy types derive an abstract machine similar to the one we have constructed in our work. The type system they consider is augmented with Jensen's strictness logic [113]. As a consequence, a computational counterpart for the typing for lambda-abstractions involves iterating through multiple *abstract values* of the formal parameter's type, which leads to the exponential complexity of the derivation algorithm. In our transition system this possible pitfall would correspond to the computation of the transition rules [lam] and [t-lam] in Figure 4.1, as they deal with processing the argument type of a function and emitting the control element *Lam*. The abstract machine-like representation allows one to coarsen the result of a type derivation by choosing different abstract domains to iterate through when the control element *Lam* is processe, thus reducing a machine's abstract state space.

A similar idea is applicable to a more recent work on a type system for security and abstract non-interference by Zanardini [216]. Non-interference refers to the possibility that two computations can be distinguished by observing some public parts of data. Types in the described system have denotational meaning and are defined in terms of abstract value domains, thereby identifying properties which cannot be distinguish in a particular domain. Parity or the sign of an integer, are the simplest examples of such abstract properties. An appropriate type system is encoded originally in terms of derivation rules, which involve iterations through possibly infinite semantic domains. From the abstract machine point of view, such an iteration would be triggered by control stack elements. It does not change the nature of type to be computed but makes it more precise depending on chosen semantic domains, which define the set of control elements. Thus, an abstract machine-like representation would give an effective way to control the precision of the type-based analysis just by redefining the meaning of appropriate control stack elements. Exploring this possibility is a direction for the future work.

Chapter 6

Conclusion and Future Work

6.1 Summary of Contributions

In program analysis, one always has to distinguish between an analysis as its formal definition and a program that implements it. In this work, we proposed a methodology to bridge this gap by using the inter-derivational method due Reynolds [174] and Danvy et al. [52]. As an example, two non-standard implementations of a traditional type checking algorithm were considered: one in the form of recursive descent and another in the form of Landin's SECD machine. The correspondence between these two models was provided by the construction and inter-derivation of their computational counterparts. Through a series of behaviour-preserving program transformations we have shown that both models are computationally equivalent. Starting from one particular traversal strategy, a family of algorithms was derived. All of them implement the *same* traversal strategy, but exhibit different computational properties. To the best of our knowledge, this is the first application of the study of the functional correspondence semantics to type systems. The result is a step towards reusing different computational models for compositional program analyses, such that the equivalence of the models is correct by construction.

6.2 Future work

6.2.1 Handling type system evolution

In further research we are going to address scalability issues of the described approach employing functional correspondence. In the presence of various pluggable type systems one may want to augment the typing rules or add new ones, so the resulting abstract machine will change as well. The natural question is how to reflect these changes incrementally without going again through the whole chain of transformations. We also leave a comparison of different implementations of a type checker with respect to performance to the future work.

6.2.2 Incorporating term substitutions

In Chapter 3, we started our derivation from a reduction-based small-step semantics with closures. In contrast, the original semantics by Kuan [122] used reduction-based semantics with capture-avoiding substitutions. For the sake of brevity, in the present work we have referred on the correspondence between the applicative order version of Curien's calculus, which uses closures, and a calculus with capture-avoiding substitutions [17]. However, we believe that this discrepancy could be also avoided by applying the functional correspondence to a substitution procedure as described by Pierce [164, Chapter 7] yielding a form of context/closure decomposition we started from in Section 3.3.4. *Shifting* (i.e., alpha-renaming) of variables would be then unnecessary, since substitution does not need to be capture-avoiding, as variable and type names cohabit in different semantic spaces. We leave incorporation of this derivation to the future work.

6.2.3 Relation to attribute grammars

The relation between attribute grammar approaches [73, 118] and the described transformations described in this dissertation is another interesting topic of discussion. Since attributes are functions from AST nodes to attribute values, type checking can be represented as a computation of such attribute values. However, the approach described deals with an *eager* semantics of type checking whereas practical attribute grammars perform the value computation *lazily*. The possible way to unify these two approaches is to derive a call-by-need semantics for type checking in the spirit of the recent work by Danvy et al. [61].

6.2.4 Application of functional correspondence to other semantic formalisms

Rewriting logic semantics for type inference Meseguer and Roşu proposed rewriting logic semantics (RLS) as a programing language definitional framework that unifies operational and algebraic denotational semantics [134]. Later, Ellison et al. applied the term-rewriting mechanism for define polymorphic type inference algorithm à la Hindley-Milner [75]. The authors focus on performance aspects of the derived interpreter and mention its similarity to Kuan et al's reduction semantics [123]. However, to the best of our knowledge, no formal correspondence between execution of a formalism in RLS and reduction semantics is established so far, which makes rewriting logic an exciting candidate to be included into the chain of inter-derivable semantics artifacts.

Semantics of languages with aspects In this work, we have demonstrated an application of functional correspondence to the type checking semantics. However, the degree of applicability of the technique is much wider. Another possible client of the approach would be a semantics for Aspect-Oriented Programming (AOP) [116]. One of the most elaborate semantics so far is the one suggested by Dutchyn in his PhD dissertation [72]. The suggested formalism is based on the well-known CEK machine with the form of a first-order control operators. *Advice*, a crucial concept of AOP, are implemented as first-order continuations stored in a separate component of the machine state. The semantics used later to derive a model of execution levels in AOP, which helped successfully resolve the long-standing problem of aspect interference [199]. We believe that taking the Dutchyn's original semantics and applying the discussed transformations would help to obtain a compositional natural or denotational semantics for higher-order aspect-oriented languages and give a solid basis for constructing static analysis form programs with aspects.

6.2.5 Mechanization of transformations

A natural further step in the line of research we presented is to provide mechanized support for functional correspondence. The PLT Redex framework [79] is one of the most promising candidates for the role of such a platform, however, so far it provides only a limited amount of facilities for semantic formalisms, focusing mainly on context-based reduction semantics. Planning to incorporate the functional correspondence approach to the PLT Redex environment, one should expect a lot of complication with the macro-expansion system of Racket, to which PLT Redex heavily relies.

Another promising direction is a mechanization of the meta-theory behind the functional correspondence in a proof assistant, such as Coq [15] or Agda [154]. Some work is done in this direction, for instance, the formalization of the CPS transformation [67] and refocusing [187]. In the recent work, Swierstra provided an implementation of a derivation of an executable Krivine abstract machine [120] from a small step interpreter for the simply typed lambda calculus in the dependently typed programming language Agda, following the functional correspondence technique [197]. The approach contains multiple useful techniques known in the dependently-typed programming community, for instance, views [132] and Bove-Capretta transformation of non-structurally recursive definitions [22]. However, a general framework incorporating the formalization of all transformations from the functional correspondence toolchain is still missing.

Part II

A Gradual Type System for Object Ownership

Chapter 7

Introduction and Problem Statement

> In practice, programmers are more than willing to add type annotations to guide the type inference engine, and to document their code. However, the choice of just what annotations are required, and what changes are required in the type system, has been an ongoing topic of research.
>
> SIMON PEYTON JONES

Type systems for ownership in object-oriented languages provide a declarative way to statically enforce a notion of object encapsulation in object-oriented programs. Object ownership ensures that objects cannot escape from the *scope* of the object or collection of objects that *own* them. Variants of ownership types allow a program to enjoy such computational properties as data race-freedom [24], disjointness of effects [33], various confinement properties [205] and effective memory management [25]. Ownership types also enable modular reasoning using knowledge about aliasing [150].

7.1 The Problem: Making Ownership Types Practical

There are several obstacles to the adoption of ownership types. The first one is verbosity. A programmer should declare explicitly the ownership structure which instances of different classes are expected to have. Moreover, a number of annotations

should be provided for fields, parameters, method results and local variables in order to preserve the information about the ownership structure in the whole program and perform an adequate type checking. One way to overcome the verbosity problem is to omit annotations and use type inference instead. However, unlike traditional type systems, ownership annotations are mostly design-driven, thus full inference of ownership types is not particularly useful, since a correct, trivial (but not useful) ownership typing always exists [69]. Therefore, inference is only practically applicable when some annotations are already provided to indicate the programmer's intent. But even in the case of partially-annotated programs, ownership type inference tends to produce an excessive amount of inferred annotations [147] or imprecise results due to the conservatism of the underlying analysis [143]. A second obstacle is that ownership types are often too rigid and restrictive to capture the dynamic evolution of an object graph in real applications, and in some cases the constraints imposed need to be relaxed.

7.2 The Method: Gradual Types

Adding ownership annotations into the code is similar to the migration from untyped to typed code, a topic of much research nowadays [86, 109, 188, 189]. Complete absence of types facilitates the fast prototyping and rapid evolution of a system, so one might need to introduce types into the code only when the demands for reliability and performance of the program are established. Ownership types provide more fine-grained safety guarantees. In this respect refactoring a program to employ them can be considered as a migration from typed to "even more typed" code. This observation leads to the idea of applying a *gradual* approach for an incremental migration.

Considering a type as a *set of data and operations allowed on this data*, one may wonder what these allowed operations are. For instance, applying an integer value as a function to some argument or calling a non-declared method of an object are not allowed operations, which are checked by traditional type systems. More advanced type systems help to check the programs for even more non-trivial but incorrect behaviour, such as null-pointer errors [77] or incorrect object initialization [171]. However, objects as data structures *do not carry* information about ownership: it should be *declared* by the programmer, and subsequently *checked* by the compiler. Hence, the role of type systems for ownership in static program analysis is twofold: they provide both mechanisms for *declaring* an invariant by augmenting a data structure with additional information and for *checking* the declared invariant.

This separation of type annotations into "declarations" and "checking-helpers" makes it possible to provide a gradual approach for ownership types and ownership

invariants.[1] A chosen ownership policy states the minimal amount of "declaration" annotations to indicate the intention of the programmer with respect to the safety invariant, ensured by this policy. The rest of annotations is considered as "helpers", which are optional, thus, can be statically omitted, so the compiler will insert necessary dynamic casts and checks. Certainly, a fully-annotated program will statically ensure the desired property, just like the traditionally well-typed programs "do not get stuck" [212].

7.3 Intuition behind Gradual Ownership Types

This section gives the essence of ownership types enforcing the *owners-as-dominators* policy (OAD) and provides some intuition on the gradualization of the type system from two perspectives: one of a working programmer and another of a programming language theoretician.

Ownership types are based on a *nesting* relation on objects ($\prec$). At run-time, each object o has an *owner*, which is another object o', such that $o \prec o'$. Nesting is a tree-shaped partial order on objects with greatest element **`world`**. The OAD invariant can be informally stated as follows in terms of a program's run-time object graph:

> Given an object o and its owner o', every path in the object graph from the program roots along object fields that ends in o, contains o'.

This means, there are *no* object fields referring to o that bypass o'. This means that one object *cannot* refer to a second object directly as a field, unless the first object is *inside* the second object's owner. This kind of ownership is called *deep*, since the nesting is transitive. Its counterpart is *shallow* ownership: the access to objects is controlled without enforcing an object graph property such as owners-as-dominators.

7.3.1 Gradual ownership types: a programmer's view

From the programmer's perspective, the object-level encapsulation property described by the OAD invariant is something the implementor of a container class would normally like to enforce. The intention would be to limit the scope of instances of auxiliary classes relevant to the implementation and make them to be strictly *within*

[1] The same separation of roles of type annotations is also typical for other domain-specific type systems, such as for security [151].

the instance of the container. The following partial implementation of the class `List` in Java [95] uses instances of the class `Link` to keep the traditional single-linked structure:

```
class List {
  Link head;
  void add(Data d) {
    head = new Link(head, d);
  }
  //  other methods
}
class Link {
  Link next;
  Data data;
  Link(Link next, Data data) {
    this.next = next; this.data = data;
  }
}
```

In order to indicate the fact that instances of the class link can be confined within some other instance object (i.e., have an owner), this owner should be mentioned explicitly in the header of the class `Link`. The class then becomes ownership-parametrized and is defined now as **`class`** `Link<owner>`.

The next step of the migration is to specialize the instance of `Link`, created within the method `add()` of `List`, i.e, indicate the fact that *this* instance of `List` is the owner of the newly instantiated `Link`. This is done by adding one more annotation to the allocation site: **`new`** `Link<`**`this`**`>(head, d)`.

So far, all the work has been done by a programmer, so the reader might wonder, where the gradual typing actually is. In fact, this amount of annotations is the *only* work the programmer should do to indicate her intentions about *ownership structure* in the considered program, namely, stating which classes *can* be owned and which particular instances *should* be owned. These annotations provide sufficient information for the compiler to figure out what checks should be performed at runtime in order to ensure the owners-as-dominators invariant. Of course, the programmer can specify more fine-grained constraints by adding more annotations into the code. For instance, to ensure that an instance of `Link` is always followed by a link with the same owner, one should add following annotations (greyed), making the invariant stronger:

```
class Link<owner> {
  Link <owner> next; Data data;
  Link(Link <owner> next, Data data) {
    this.next = next; this.data = data;
  }
}
```

In the approach we take, annotating both fields and method parameters is *not necessary*. However, doing so reduces the amount of runtime assertions. For instance, had we omitted the annotation of the field `next`, each assignment to it would cause an extra check of the invariant. Otherwise one could assign to `next` an instance of `Link`, owned, for instance, by another list. Other possible views to gradualization of the ownership type system can be taken, for instance, allowing one to omit class and instance annotations, but provide field annotations instead as "ownership interfaces". We we consider these possibilities as well as their drawbacks in Chapter 10.

Summarizing, Figure 7.1 gives the full code of our running example: the class `List` using ownership types [33]. The latter implementation of the class carries two ownership parameters: `owner` and `data`. The first parameter, `owner`, refers to the `List` instance's immediate, or *primary*, owner. The second parameter, `data`, refers, by conventions of the type system, to some object *outside* or equal to `owner`. As usual, **`this`** refers to the current instance itself. The same reasoning is applicable to two auxiliary classes, `Link` and `Iterator`. In the `List`'s method `add()`, the programmer indicates, by creating an instance of the class `Link` with owners **`this`** and `data` respectively, that this particular instance of `Link` is nested within its creator instance `List` and the content of the link can be accessed only through the owner referred to as `data` in `List`. The same is true for the instance of the class `Iterator`. Figure 7.2 presents an informal diagram illustrating the desired dynamic heap topology and encapsulation boundaries.

7.3.2 Gradual ownership types: a semanticist's view

A programming language theoretician may wonder, which part of the annotations in the type system does actually "do the trick" when defining the desired the ownership relation, and which just serves in order to handle assigned types correctly.

Provided and optional type annotations One can notice that all the ownership information required to describe the topology for the list example as on Figure 7.2, can be provided by only five annotations. Three class parametrizations name the owners of the class instances and two allocation sites provide concrete owners for created objects. These annotations, emphasized by framed boxes in Figure 7.1, declare the information about ownership structure and nesting of objects involved (i.e., **`this`** $\prec$ `owner` $\prec$ `data` $\prec$ **`world`**) and define the owners of new instances. The remaining annotations, greyed in the code, serve as constraints and "propagate" ownership information through the program, as mutable variables and fields are traditionally annotated with types to keep information about objects they point to.

We require the first kind of annotations to be explicitly specified, because it

```
class List <owner, data> {
  Link <this, data> head;
  void add(Data <data> d) {
    head = new Link <this, data> (head, d);
  }
  Iterator <this, data> makeIterator() {
    return new Iterator <this, data> (head);
  }
}
class Link <owner,data> {
  Link <owner, data> next;
  Data <data> data;
  Link(Link <owner, data> next, Data <data> data) {
    this.next = next; this.data = data;
  }
}
class Iterator <owner, data> {
  Link <owner, data> current;
  Iterator(Link <owner, data> first) {
    current = first;
  }
  void next() { current = current.next; }
  Data <data> elem() { return current.data; }
  boolean done() { return (current == null); }
}

// An arbitrary class to represent data elements
class Data <dataOwner> {
  Object <world> myArbitraryField;
}


Object listOwner = ...
Object dataOwner = ...

Data <dataOwner> d1 = new Data <dataOwner> ();
Data <dataOwner> d2 = new Data <dataOwner> ();

List <listOwner, dataOwner> list = new List <listOwner, dataOwner> ();
list.add(d1);
list.add(d2);

Iterator <list, dataOwner> iter = list.makeIterator();
Data <dataOwner> fetched = iter.elem();

// Illegal reference
fetched.myArbitraryField = list.head;
```

Figure 7.1: A motivating example and the design intention: a list and its iterator code with structural (boxed) and constraint (greyed) ownership annotations.

(a) reflects the programmer's intentions with respect to the invariant, and,

(b) enables a simpler implementation of run-time dynamic checking, i.e., no ownership information needs to be inferred dynamically (see Chapter 10 for a detailed discussion).

Alternative approaches are discussed in Chapter 10 of this dissertation.

Type casts and boundary checks The runtime checking of conformance of an object's ownership structure to its expected structure, imposed by the type, is performed at run-time via dynamic *type casts*. This technique is typical for gradual approaches: when an untyped value is coerced to a typed value, a dynamic check is performed to ensure that the further interactions through this particular reference conform to the target's type contract, in this setting, its ownership type. However, the preservation of the OAD invariant requires not only conformance of *actual* and *expected* types, but also checking that the nesting constraints are preserved—this information is lost when ownership information is lost.

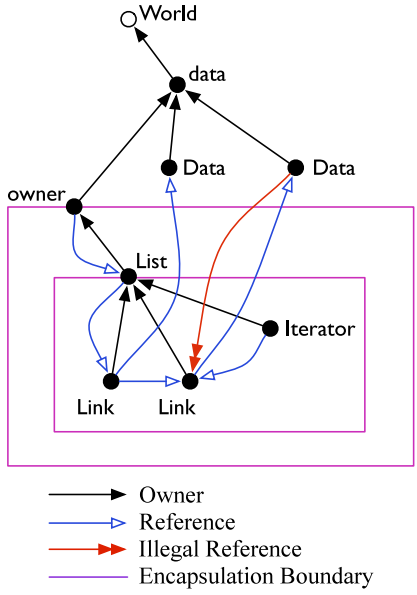

Figure 7.2: A desired heap topology for the program in Figure 7.1.

The only place where the owners-as-dominators invariant can actually be broken is by a bad field assignment, which makes field assignments good candidates for extra run-time checks. Let us consider the following code:

```
class D<owner> {
  D myD;
  ...
}

D<q> otherD = ...;
D<p> d = new D<p>();
d.myD = otherD;
```

The correctness requirement for such an assignment demands that p $\prec$ q. However, the declaration of the field myD lacks ownership information, so there is a chance that the OAD invariant will be violated since the type of myD may no longer impose any nesting between p and q. This is a sort of contract that should be checked dynamically. We call these *boundary checks*.

One can notice that dynamic type casts operate with objects' ownership structures, whereas boundary checks traverse a part of the heap and, therefore, are significantly more expensive in terms of execution overhead. However, performing type casts before boundary checks might help to avoid most of them, since after the check we gain some extra knowledge of an object's structure. This observation leads us to a two-staged, typed-directed transformation, where each stage uses the available type information to insert one sort of check: for type conformance and nesting, respectively. In the following chapters we will develop a staged algorithm for the correct translation. The first pass will insert dynamic casts and the second will handle possible OAD violations by inserting boundary checks.

Formal, unknown and dependent owners An important part of the ownership type system is the static representation of owners. The example in Figure 7.1 demonstrated one usage of ownership class parameters. The following example exhibits the concept of *dynamic aliasing* [33], which employs local final variables as local owners:

```
final List<p, world> list = new List<p, world>();
Iterator<list, world> iter = list.makeIterator();
```

Variable list denotes the owner of the iterator iter. When list goes out of scope, the type Iterator<list, **world**> and other types containing owner list become illegal.

Following gradual types we introduce a notion of the special *unknown owner* "?". Types annotated with "?" in a gradually-typed language defer the checking of types to run-time via checks inserted by the compiler. In our system, types with no annotations are just syntactic sugar for types with all ownership annotations unknown, e.g., List $\equiv$ List<?,?>. The following code gives the essence of unknown owners:

```
List list;  // ≡ List<?,?>
list = new List<p, world>();
list = new List<this, world>();
List<p, world> newList = list; // inserted cast (List<p, world>)list
```

The first two assignments are valid since the type of list does not specify which objects must own the instance referred by the variable. The last assignment

```
1  class E<P> { D myD = ... }
2
3  class D<owner> {
4    E<owner> e;
5    void use(D<owner> arg) { ... }
6    void exploit(E<owner> arg) { this.e = arg; }
7    void test(E e) {
8      final D d = e.myD; // implicitly, d: D<d^D.owner>
9      d.use(d); // type refinement, but no type cast required
10     d.exploit(e); // type refinement, type cast required
11   }
12 }
```

Figure 7.3: Dependent owners in action.

is valid too; however, it requires a dynamic cast, due to the type refinement `List<?, ?>` ⇒ `List<p, world>` to make sure that the owners of `list` match the specification of `newList`.

Information lost due to unknown owners can be partially regained by tracking dependencies between immutable references and owners of objects they refer to. For this purpose we introduce the notion of *dependent owners* that record the origin of some owner arguments, allowing one to check them for equality without knowledge about how they are nested.

Figure 7.3 provides some intuition about dependent owners. Class `E` declares a field of type `D`. However, information about the owner of the object referred to by field `myD` is lost due to the missing ownership annotation in the field declaration at line 1. As a consequence, the owner of variable `d` at line 9 is unknown. Nevertheless, since `d` is final, one can see that the owner of the object referred by `d` is the same as the one expected as of a parameter of the instance method `d.use()`. This knowledge is preserved by assigning the type $\texttt{D<d}^{\texttt{D.owner}}\texttt{>}$ to variable `d`. This should be read as "`d` has the type `D` and the owner of the object referred to by `d` is locally denoted as $\texttt{d}^{\texttt{D.owner}}$". The superscript `D.owner` refers to the particular ownership parameter of the statically known type `D`. Thus, by equality of owners, no extra dynamic check is required at line 9. Still, the owner of `e` remains unknown, so the method call `d.exploit()` at line 10 is potentially dangerous due to type refinement, and therefore the cast `E<?>` ⇒ $\texttt{E<d}^{\texttt{D.owner}}\texttt{>}$ is required. Without dependent owners we would lose precision when checking types.[2]

[2]We have chosen the term "dependent owners" because of similarity of the idea to the notion of *path-dependent types* [157]—the value of the owner depends on the value of an object.

7.4 Main Contributions

This part of the dissertation contributes Gradual Ownership Types, a framework for the gradual migration of programs in Java-like object-oriented languages to a richer type system in the spirit of the one by Clarke and Drossopoulou [33], which ensures the owners-as-dominators encapsulation invariant. We investigate formal aspects of gradual ownership types, discuss the implementation and evaluate the approach on a well-studied codebase. Overall, our work makes the following contributions.

A formalization of a gradual ownership type system and a type-directed compilation for a Java-like language

Our main contribution in this part of the dissertation is a formalisation of gradual ownership type system in the setting of a toy Java-like language, similar to Featherweight Java [108], but featuring an explicit mutable heap. The core calculus is based on the ownership type system of Clarke and Drossopoulou [33], which ensures the owners-as-dominators invariant. The developed type system establishes a version of a "relaxed" typing relation that introduces the notion of *unknown* owners and allows the compiler to move parts of invariant checking from compilation to the run-time phase. We also formalize a type-directed compilation procedure, which inserts dynamic checks into the code whenever a type checker fails to ensure the ownership invariant preservation statically. Finally, the translation is proven to be sound with respect to the described gradual system (i.e., every gradually well-typed program can be compiled into a well-typed program with dynamic checks), and a proof of the ownership invariant preservation in a well-typed program with checks is provided.

The overall system enables the migration from ownership-unaware to ownership-annotated code. It is expressive enough to investigate the concepts of interest and is close enough to a real language to guide the implementation.

A translating compiler for the gradual ownership types

Our second contribution is a proof-of-concept implementation of a translating compiler for full Java 1.4 that supports gradual ownership types and provides hints for a smooth program migration. The implementation is made as an extension of the JastAddJ extensible Java compiler [74] using the JastAdd framework, which based on the extensive use of attribute grammars [73]. The compiler itself and its testsuite with numerous small examples of programs with gradual ownership types are freely available on GitHub for extension and experiments:

http://github.com/ilyasergey/Gradual-Ownership

A report on program migration using gradual ownership types

Our third contribution is an evaluation of the developed compiler for Java with gradual ownership types in the scope of the Java Collection Library. We report on migrating several collection classes starting from the minimal amount of type annotations in order to ensure the invariant and using compiler support for debugging and gradual migration. We provide an example of a class that we were able to instrument completely with owner annotations and, therefore, reach full static guaranties of the designed invariant preservation. We also report on problems when migrating some classes due to limitations in the expressiveness of the formalism, which can be remedied in the future work. Finally, we observe a case of a class that might experience a sort of undesired behaviour, when an owned object "leaks" its owner's scope.

A discussion on gradualization of type systems for object ownership

Our additional contribution is a discussion on extensions of the presented method in the context of alternative type systems for strong encapsulation in object-oriented programs. Although the described approach of "gradualization" is presented in the scope of the owners-as-dominator invariant, we argue that it is idiomatic and sketch the ways it can be applied to some other ownership type systems. We also initiate a discussion on possible design choices when implementing gradual ownership types, accounting to both expressiveness and efficiency.

7.5 Outline

Part II of the dissertation describes the research based on a combination of the following research papers and technical reports:

- Ilya Sergey and Dave Clarke. Gradual Ownership Types. In Helmut Seidl, editor, proceedings of the *21th European Symposium on Programming (ESOP 2012)*, volume 7211 of *Lecture Notes in Computer Science*, pages 579–599. 24 March – 1 April 2012. Tallinn, Estonia. Springer.

- Ilya Sergey and Dave Clarke. Gradual Ownership Types. *CW Reports, volume CW613*, 43 pages, Department of Computer Science, KU Leuven, Leuven, Belgium, December 2011.
- Ilya Sergey and Dave Clarke. Towards Gradual Ownership Types. In *International Workshop on Aliasing, Confinement and Ownership (IWACO 2011)*. 25 July 2011. Lancaster, UK.

The remainder of this part is structured as follows.

Chapter 8 – Calculus of Gradual Ownership Types

The chapter presents $\mathsf{JO}_?$, a core calculus of gradual ownership types and its extended version $\mathsf{JO}_?^+$, accounting for dynamic type casts and boundary checks. First, the syntax and static semantics of $\mathsf{JO}_?$ are described. Second, the intuition behind the dynamic checks for the owners-as-dominators invariant is recalled and a type-directed translation from $\mathsf{JO}_?$ to $\mathsf{JO}_?^+$ is presented along with enhanced typing rules for $\mathsf{JO}_?^+$. An operational semantics of $\mathsf{JO}_?^+$ is defined in the form of an abstract CEK-machine with an explicit heap component. Finally, the subject reduction and the owners-as-dominator invariant preservation theorems are formulated and proven. The last results of the chapter state that a well-typed program in $\mathsf{JO}_?$ can be always compiled to a well-typed program in $\mathsf{JO}_?^+$ and execution of the latter will not violate the owners-as-dominator invariant.

Chapter 9 – Implementation and Evaluation

In this chapter, we describe the main design decisions taken while implementing the gradual ownership types framework as an extension of the JastAddJ extensible Java compiler for Java 1.4. In particular, we discuss issues that arose during the implementation of dependent owners, supporting typical object-oriented features: anonymous and inner classes, and virtual method overriding. We also provide a report on the experience of migrating some container implementations from the Java Collection Library to support ownership type annotations. We discuss the performance overhead and some idiomatic cases when the described approach cannot be applied "as is" due to limitations in its expressiveness.

Chapter 10 – Discussion and Related Work

In this chapter, we start with a discussion on extending the framework of gradual ownership types for different existing ownership policies from the literature. We

continue with an overview of possible design choices of the implementation in order to trade the flexibility of the type system to the runtime performance and vise versa. We then proceed with an overview of related work, giving a birds-eye view on modern hybrid type systems in mainstream programming languages and focusing mainly on techniques for ownership inference as a main competitor to the approach described in the present work.

Chapter 11 – Conclusion and Future Work

In the last chapter of this part we reflect on the contributions of the work, and summarize main challenges to be overcome in future investigations, mainly, with respect to modularity and better assistance for incremental code migration. Finally, we discuss possible applications of the presented technique in practice.

Chapter 8

A Calculus of Gradual Ownership Types

> Every time programming language theoreticians have troubles understanding what is going on, they invent a new calculus.
>
> NEEL KRISHNASWAMI

In this chapter, we discuss theoretical aspects of gradual ownership types. In Section 8.1, we introduce $\mathsf{JO}_?$, a core Featherweight Java-like language with gradual ownership types, and provide its static semantics. In Section 8.2, we define the type-directed compilation scheme in the form of a translation from $\mathsf{JO}_?$ to an extended language $\mathsf{JO}_?^+$, augmented with type casts and boundary checks in order to provide dynamic guaranty of the ownership invariant. Section 8.3 presents the dynamic semantics of the language $\mathsf{JO}_?^+$. Finally, Section 8.4 states main theorems about the formalism: subject reduction, invariant preservation in $\mathsf{JO}_?^+$ and static type safety of $\mathsf{JO}_?$ with respect to the compilation procedure.

8.1 The language $\mathsf{JO}_?$

To investigate the meta-theory of gradual ownership types, we define $\mathsf{JO}_?$, an imperative Java-like language, extended with ownership types, and unknown and dependent owners, based on the system JOE_1 by Clarke and Drossopoulou [33].

Programs	P	::=	$class_{j\in 1..m}$
Class declarations	$class$	::=	$\texttt{class}\ c\langle\alpha_{i\in 1..n}\rangle\ \texttt{extends}\ c'\langle p_{j\in 1..n'}\rangle$ $\{fd_{k\in 1..m};\ meth_{l\in 1..u}\}$
Field declarations	fd	::=	$t\ f$
Method declarations	$meth$	::=	$t\ m(t\ x)\ \{e\}$
Expressions	e	::=	$x \mid \texttt{let}\ x = b\ \texttt{in}\ e \mid v$
Computations	b	::=	$x.f \mid x.f = x \mid x.m(x) \mid$ $\texttt{new}\ c\langle p_{i\in 1..n}\rangle \mid \texttt{null}$
Values	v	::=	$\iota \mid \texttt{null}$
Run-time owners	k	::=	$\texttt{world} \mid \iota$
Owners	p,q	::=	$x \mid \texttt{this} \mid k \mid ? \mid \alpha$
Types	t,s	::=	$c\langle p_{i\in 1..n}\rangle$
Objects	o	::=	$\langle c\langle k_{i\in 1..n}\rangle, \overline{f \mapsto v}\rangle$
Typing environments	E	::=	$\emptyset \mid E, x:t \mid E, \iota:t \mid E, p \prec p'$
Bindings	B	::=	$\emptyset \mid B, \alpha = k \mid B, x = v$
Heaps	H	::=	$\overline{\iota \mapsto o}$
Variables	$x,y,z,\texttt{this}$		
Heap locations	ι		
Formal owners	α		
Dependent owners	$\boxed{x^{c.i}}$		
Unknown owners	$?$		

Figure 8.1: Syntax of JO?. Runtime syntax elements are emphasized by grey boxes. Boxed elements denote syntactic elements, inserted by a compiler, not by a programmer.

8.1.1 Syntax

Programs, classes and expressions Figure 8.1 provides the full syntax of JO?. Entities used at runtime only are in grey. We also emphasize administrative entities, used by the translating compiler, but not by the programmer, using framed boxes. A program in JO? is a collection of classes. A class definition

$$\texttt{class}\ c\langle\alpha_{i\in 1..n}\rangle\ \texttt{extends}\ c'\langle p_{j\in 1..n'}\rangle\{fd_{k\in 1..m};\ meth_{l\in 1..p}\}$$

describes a class named c, parametrized by the ownership parameters $\alpha_{i\in 1..n}$ with the superclass c', whose ownership parameters are instantiated with $p_{j\in 1..n'}$.[1]

The class contains fields $fd_{k\in 1..m}$ and methods $meth_{l\in 1..u}$; a field is defined by a type annotation and a field name: $t\ f$; a method is defined by its return type, name, formal parameter signature, and body expression: $t\ m(t\ x)\ \{e\}$. Methods have only one parameter for simplicity. Expressions in $\mathsf{JO}_?$ are in the administrative normal form (ANF) [87], i.e., results of all intermediate atomic computations are named and assigned to the immutable variables. Local variables can be used as owners, as long as they do not escape the scope of their local stack frame [33]. This is a weakening of the original formulation of the owners-as-dominators invariant, which also considered local stack frame references as possible edges of an object graph preventing one, for instance, from using iterators as in Figure 7.1 [36].

Atomic computations include field lookup, field update, method call, object creation and `null`.

Types and owners A type $c\langle p_{i\in 1..n}\rangle$ consists of a class name c and a vector of ownership arguments $p_{i\in 1..n}$.

Owners are represented syntactically by owner and term variables (α and x, respectively), dependent owners and *run-time* owners such as **`world`** and heap locations (i.e, run-time object identifiers). Run-time values in owner positions are introduced into the source language to simplify the progress/preservation formulation and proofs. $x^{c.i}$ denotes the dependent owner corresponding to the i-th ownership parameter of the object referred to by the term variable x, whose statically known class type is c. Dependent owners are not supposed to be specified by the programmer. Instead, they are inferred by the compiler. For convenience, we often use an alternative notation $c\langle\sigma\rangle$ for a type $c\langle p_{i\in 1..n}\rangle$, assuming σ to be a substitution $\{\alpha_i \mapsto p_i \mid i \in 1..\mathsf{arity}(c)\}$, where α_i are formal ownership parameters of the class c.

To distinguish between different kinds of owners when checking, the well-formedness of types, we introduce several helper functions (Figure 8.2).

Objects and heaps In addition to having the class name and field values, an object also has a binding for its owner parameters, either `world` or some non-null heap locations. A heap H is a partially defined map from locations to objects.

[1] More expressive possibilities exist in the literature, for example, by allowing the programmer to declare the expected relationship the between owner parameters of a class [37].

$$
\begin{array}{rcl}
\mathsf{defined}(p) & \triangleq & p \neq ? \text{ and } p \neq x^{c.i} \\
\mathsf{undefined}(p) & \triangleq & \neg\mathsf{defined}(p) \\
\mathsf{actual}(p) & \triangleq & p = \texttt{world} \text{ and } p = \iota \text{ for some } \iota \\
\mathsf{arity}(c) & \triangleq & n, \text{ s.t. } \texttt{class}\ c\langle \alpha_{i\in 1..n}\rangle \in P \\
 & & \\
\mathsf{owner}(c\langle\rangle) & \triangleq & \texttt{world} \\
\mathsf{owner}(c\langle p_{i\in 1..n}\rangle) & \triangleq & p_1, \text{ where } n > 0 \\
\mathsf{owner}_j(c\langle p_{i\in 1..n}\rangle) & \triangleq & p_j, \text{ where } 0 < j \leq n \\
\mathsf{owners}(c\langle p_{i\in 1..n}\rangle) & \triangleq & p_1 \ldots p_n
\end{array}
$$

Figure 8.2: Helper functions of JO?.

8.1.2 Typing environments and owners

A typing environment E binds variables and heap locations to types and defines ordering assumptions on owners in terms the nesting relation $\prec$. The bindings B map formal owners to run-time owners and variables to values.

The dynamic semantics is defined in Section 8.3 in terms of an explicit binding of free variables, rather than via substitution. The presence of binding environment in the typing judgements of the form $E;B \vdash \mathfrak{F}$ for some succedent $\mathfrak{F}$ does not affect the static semantics of JO?, but we will need it to establish equalities between typing environments and dynamic bindings in the proof of the type preservation theorem. To avoid duplicating work, we include a binding list in the assumption set of most judgements.

A typing environment E is *well-formed* if $\prec$ is antisymmetric on $\mathrm{dom}(E)$, i.e, the environment does not introduce cycles in ownership. The definition of well-formed environment-binding pairs ($E;B \vdash \diamond$) will be described in detail in Section 8.4. Informally, the pair $E;B$ enables owners and types in E to be used modulo equalities in the run-time binding environment B. To keep the presentation tractable, we avoid explicitly stating rules dealing with such equalities.

The *well-formed owner* relation ($E;B \vdash p$) is shown in Figure 8.3. The rules (OWN-DEPENDENT) and (OWN-?) are novel for the gradual type system.

It is important to notice that a dependent owner is well-formed only if the corresponding variable is bound by a typing environment with some class c and it requires that i does not exceed the ownership-arity of the class c (the rule (OWN-DEPENDENT)). The reason of this is the fact that dependent owners in practice are not inserted by the programmer, but are artifacts of the type-checking procedure, which

$\boxed{E;B \vdash p}$ Well-formed owners

$$\frac{E;B \vdash \diamond}{E;B \vdash \texttt{world}}\ (\textsc{own-world}) \qquad \frac{E;B \vdash \diamond \quad E;B \vdash x : t}{E;B \vdash x}\ (\textsc{own-var}) \qquad \frac{E;B \vdash \diamond \quad E;B \vdash \mathfrak{l} : t}{E;B \vdash \mathfrak{l}}\ (\textsc{own-val})$$

$$\frac{E;B \vdash \diamond}{E;B \vdash ?}\ (\textsc{own-?}) \qquad \frac{E;B \vdash x : c\langle \ldots \rangle \text{ for some class } c \quad i \in 1..\mathsf{arity}(c)}{E;B \vdash x^{c.i}}\ (\textsc{own-dependent}) \qquad \frac{E;B \vdash \diamond \quad p \prec p' \in E}{E;B \vdash p, p'}\ (\textsc{own-in})$$

$\boxed{E;B \vdash p \prec p'}$ Nested owners, $\mathsf{defined}(p)$, $\mathsf{defined}(p')$

$$\frac{p \prec p' \in E}{E;B \vdash p \prec p'}\ (\textsc{in-env}) \qquad \frac{E;B \vdash p}{E;B \vdash p \prec p}\ (\textsc{in-refl}) \qquad \frac{E;B \vdash p \prec p' \quad E;B \vdash p' \prec p''}{E;B \vdash p \prec p''}\ (\textsc{in-trans}) \qquad \frac{E;B \vdash x : t \quad p = \mathsf{owner}(t)}{E;B \vdash x \prec p}\ (\textsc{in-var})$$

$\boxed{E;B \vdash p \lesssim p'}$ Consistently-nested owners

$$\frac{E;B \vdash p \quad E;B \vdash q \quad \mathsf{undefined}(q)}{E;B \vdash p \lesssim q}\ (\textsc{sub-left}) \qquad \frac{E;B \vdash p \quad E;B \vdash q \quad \mathsf{undefined}(q)}{E;B \vdash q \lesssim p}\ (\textsc{sub-right})$$

$$\frac{E;B \vdash p \prec p'}{E;B \vdash p \lesssim p'}\ (\textsc{sub-incl}) \qquad \frac{E;B \vdash p}{E;B \vdash p \lesssim \texttt{world}}\ (\textsc{sub-world})$$

Figure 8.3: Well-formed owners and owner nesting.

$\boxed{E;B \vdash p \sim p'}$ Consistent owners

$$\frac{E;B \vdash p}{E;B \vdash p \sim p}\ (\textsc{con-refl}) \qquad \frac{E;B \vdash p \quad E;B \vdash q \quad \mathsf{undefined}(q)}{E;B \vdash q \sim p}\ (\textsc{con-right}) \qquad \frac{E;B \vdash p \quad E;B \vdash q \quad \mathsf{undefined}(q)}{E;B \vdash p \sim q}\ (\textsc{con-left})$$

$\boxed{E;B \vdash t \sim t'}$ Consistent types

$$\frac{E;B \vdash c\langle p_{i\in 1..n}\rangle \quad E;B \vdash c\langle q_{i\in 1..n}\rangle \quad p_i \sim q_i\ \forall\, i \in 1..n}{E;B \vdash c\langle p_{i\in 1..n}\rangle \sim c\langle q_{i\in 1..n}\rangle}\ (\textsc{con-type})$$

$\boxed{E;B \vdash t}$ Well-formed type t

$$\frac{\mathsf{arity}(c) = n \quad E;B \vdash p_1 \lesssim p_i \quad \forall i \in 1..n}{E;B \vdash c\langle p_{i\in 1..n}\rangle}\ (\textsc{g-type})$$

$\boxed{E;B \vdash t \leq t'}$ Nominal subtyping

$$\frac{E;B \vdash t}{E;B \vdash t \leq t}\ (\textsc{sub-refl}) \qquad \frac{E;B \vdash t \leq t' \quad E;B \vdash t' \leq t''}{E;B \vdash t \leq t''}\ (\textsc{sub-trans})$$

$$\frac{E;B \vdash c\langle\sigma\rangle \quad \texttt{class}\ c\langle\alpha_{i\in 1..n}\rangle\ \texttt{extends}\ c'\langle r_{i\in 1..n'}\rangle\{\ldots\}}{E;B \vdash c\langle\sigma\rangle \leq c'\langle\sigma(r_i)_{i\in 1..n'}\rangle}\ (\textsc{sub-class})$$

$\boxed{E;B \vdash t \lesssim t'}$ Consistent subtyping

$$\frac{E;B \vdash c\langle\sigma\rangle \leq c'\langle\sigma'\rangle \quad E;B \vdash c'\langle\sigma'\rangle \sim c'\langle\sigma''\rangle}{E;B \vdash c\langle\sigma\rangle \lesssim c'\langle\sigma''\rangle}\ (\textsc{grad-sub})$$

Figure 8.4: Owner and type consistency; well-formed types; traditional and consistent subtyping.

ensures that the corresponding "base" variable x is bound in the local scope (see the rule (T-LET) in Section 8.1.5 and the definition of function fill).

The definition of the nesting relation on owners (Figure 8.3, $E;B \vdash p \prec p'$) captures only defined owners. It is then embedded into a more general *consistent nesting* relation ($E;B \vdash p \lesssim p'$), which deals also with dependent and unknown owners. Informally, no precise information about nesting can be retrieved from unknown or dependent owners. Note that $\lesssim$ is not transitive, so $E;B \vdash q \lesssim ?$ and $E;B \vdash ? \lesssim p$ do not in general imply $E;B \vdash q \lesssim p$ for any defined p and q.

8.1.3 OAD invariant, formally

To state the OAD invariant we need a definition of a heap flattening to turn a heap into a typing environment. The notation $\widehat{H}$ represents the flattening of a heap H into a typing environment $\widehat{H}$.

Definition 8.1.1 (Heap flattening)**.**

$$\widehat{H} \triangleq \{(\iota \prec o), (\iota : c\langle o, k_{i\in 2..n}\rangle) \mid (\iota \mapsto \langle c\langle o, k_{i\in 2..n}\rangle, \ldots\rangle) \in H\}$$

Definition 8.1.2 (Owners-as-Dominators Invariant [159])**.** $\mathsf{OAD}(H) \triangleq$ for all locations ι, ι' and run-time owners k,

$$\left.\begin{array}{c} H(\iota) = \langle c\langle k_{i\in 1..n}\rangle, \overline{f \mapsto v}\rangle \\ f_i \mapsto \iota' \text{ and } H(\iota') = \langle t', \ldots\rangle \\ \mathsf{owner}(t') = k \end{array}\right\} \Rightarrow \quad \widehat{H};\emptyset \vdash \iota \prec k$$

In words, if object ι references object ι' via a field, ι must be inside the owner of ι'. The definition, however, does not prevent a flattened heap $\widehat{H}$ itself from having cyclic nesting chains (i.e., $(\iota \prec \ldots \iota' \ldots \prec \iota$ for some $\iota \neq \iota'$). We address this issue in Section 8.4.1 by defining well-formed heaps.

8.1.4 Type consistency and subtyping

Types can be constructed from any class using any owner in scope (including an unknown owner "?"), as long as the correct number of arguments are supplied and the owner (the first parameter), if present, is provably consistently-inside all other parameters. The corresponding relation $E;B \vdash t$ defining well-formed types is presented in Figure 8.4 along with the definitions of consistent owners and types.

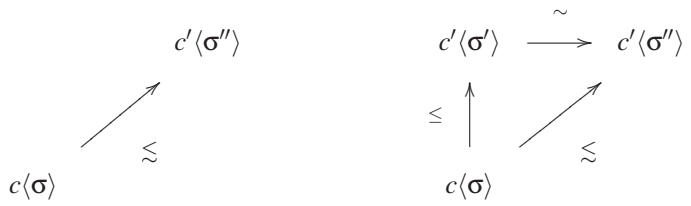

Figure 8.5: Consistent subtyping explained.

The type consistency relation answers the question: *which pairs of static types could possibly correspond to comparable run-time types?* It allows the type checker to compare types that include dependent and unknown owners. We define the type consistency relation $\sim$ on types parametrized with partially known and dependent owners via the rules in Figure 8.4 (the relation $E;B \vdash t \sim t'$).

The definition of the subtyping $E;B \vdash t \leq t'$ is standard for parametrized object-oriented type systems; ownership parameters are invariant [33]. In order to eliminate non-determinacy from the type-checking algorithms we need to construct a relation that combines two kinds of subsumption of types: type consistency and subtyping. This relation is used then in type rules whenever an implicit upcast is necessary [164].

Siek and Taha suggest a way to design such a *consistent-subtyping* relation ($\lesssim$) [189] for the calculus $\mathbf{Ob}_{<:}$ of Abadi and Cardelli [1]. If two types $t = c\langle\sigma\rangle$ and $t' = c'\langle\sigma''\rangle$ are related via the consistent-subtyping relation, i.e., $t \lesssim t'$, they can differ along both directions: the type consistency relation $\sim$ and the subtyping relation $\leq$. This is illustrated by the diagram on the left of Figure 8.5. The diagram on the right shows one possible way to define the $\lesssim$ relation through the intermediate type $c'\langle\sigma''\rangle$ such that the whole diagram commutes. According to the definition of the type-consistent relation it is easy to see that the class type of this mediator should be equal to c'. In fact, if c' is a superclass of the class c, the necessary substitution can be computed in a straightforward way by just ascending the chain of superclasses. The "upper-left mediator" is a connecting link between two types. This intuition is formalized via the rule (GRAD-SUB) in Figure 8.4.

The correspondence between a possible "bottom-right" and "upper-left" mediator from Figure 8.5 is stated by the following theorem:

Lemma 8.1.3 (Mediator switch for $\lesssim$)**.** *If $E;B \vdash t \sim t''$ and $E;B \vdash t'' \leq t'$, then $E;B \vdash t \lesssim t'$.*

Proof. "Upper-left" mediator for the definition of $\lesssim$ is built by induction on the height of the subtyping relation, type consistency is then preserved. □

Though the formal construction of the derivation for the relation $\lesssim$ might seem to be non-deterministic, it is not so. In fact, for a fixed type $c\langle\sigma\rangle$ and its super-class c' there is only one possible way to derive the corresponding super-type $c'\langle\sigma'\rangle$. This fact is due to the deterministic nature of the plain subtyping relation definition: for each pair of types t, t' at most one derivation tree for the relation $t \leq t'$ is possible.

8.1.5 Expression, method and class typing

Typing rules for computations, expressions and methods are described in Figure 8.6, following the standard approach [164]. Type rules for variables and values are standard. $m \uplus m'$ denotes the disjoint union of finite maps m and m', requiring that their domains are disjoint. σ_z is the substitution $\sigma \uplus \{\texttt{this} \mapsto z\}$ for any substitution σ. We use the mappings $\mathcal{F}_c$ and $\mathcal{MT}_c$ for retrieving types of fields and methods of a class c. In the rules (T-LET) and (METHOD), the helper function fill converts declared types with *unknown* owners to types with *dependent* owners to track owner dependencies.

$$\mathsf{fill}(x, c\langle p_{i\in 1..n}\rangle) \quad \triangleq \quad c\langle q_{i\in 1..n}\rangle, \text{ where } q_i = \begin{cases} x^{c.i} & \text{if } p_i = ? \\ p_i & \text{otherwise.} \end{cases}$$

The definitions of well-formed classes ($\vdash c$) and programs ($\vdash P; e$) are standard and are present in Figure 8.7. The class `Object` is located on the top of class hierarchy and it has only one owner parameter. A program is well-formed ($E \vdash P; e$) if its constituent classes are well-formed and its trailing, *main*, expression is well-typed.

8.2 Type-directed translation: the language $\mathsf{JO}_?^+$

This section describes the type-based translation of programs in $\mathsf{JO}_?$ to an extended language, $\mathsf{JO}_?^+$ with run-time checks, preserving the OAD invariant dynamically in the case where the provided static type annotations are not sufficient for compile-time reasoning.

$\boxed{E;B \vdash b : t}$ Well-typed computation (under $\vdash$)

$$\frac{E;B \vdash c\langle p_{i\in 1..n}\rangle \quad \mathsf{defined}(p_i)\ \ \forall i \in 1..n}{E;B \vdash \texttt{new}\ c\langle p_{i\in 1..n}\rangle : c\langle p_{i\in 1..n}\rangle}\ \text{(T-NEW)}$$

$$\frac{E;B \vdash z : c\langle\sigma\rangle \quad \mathcal{F}_c(f) = t}{E;B \vdash z.f : \sigma_z(t)}\ \text{(T-LKP)}$$

$$\frac{E;B \vdash z : c\langle\sigma\rangle \quad \mathcal{F}_c(f) = t \quad E;B \vdash y : s \quad E;B \vdash s \lesssim \sigma_z(t)}{E;B \vdash z.f = y : \sigma_z(t)}\ \text{(T-UPD)}$$

$$\frac{E;B \vdash y : s \quad \mathcal{M}\mathcal{T}_c(m) = (y', t \to t') \quad E;B \vdash z : c\langle\sigma\rangle \quad E;B \vdash s \lesssim \sigma_z(t) \quad \sigma' \equiv \sigma \uplus \{y' \mapsto y\}}{E;B \vdash z.m(y) : \sigma'_z(t')}\ \text{(T-CALL)}$$

$\boxed{E;B \vdash e : t}$ Well-typed expression

$$\frac{E;B \vdash \diamond \quad x : t \in E}{E;B \vdash x : t}\ \text{(T-VAR)}$$

$$\frac{E;B \vdash b : t \quad E, x : \mathsf{fill}(x,t);B \vdash e : s}{E;B \vdash \texttt{let}\ x = b\ \texttt{in}\ e : s}\ \text{(T-LET)}$$

$$\frac{E;B \vdash \diamond \quad \mathfrak{l} : t \in E}{E;B \vdash \mathfrak{l} : t}\ \text{(T-VAL)}$$

$$\frac{E;B \vdash t}{E;B \vdash \texttt{null} : t}\ \text{(T-NULL)}$$

$\boxed{E \vdash t'\ m(t\ y)\{e\}}$ Well-typed method (under $\vdash$)

$$\frac{E, y : \mathsf{fill}(y,t) \vdash e : s \quad E \vdash s \lesssim t'}{E \vdash t'\ m(t\ y)\{e\}}\ \text{(METHOD)}$$

Figure 8.6: Typing rules of JO$_?$ for computations, expressions and methods. Greyed parts mark explicit consistent-subtyping checks that may lead to the insertion of dynamic checks.

$\boxed{E;B \vdash c}$ Well-typed class

(CLASS-OBJECT)

$$\frac{}{\vdash \texttt{class Object}\langle\alpha_1\rangle\{\}}$$

(CLASS)

$$\frac{\begin{array}{c} E \equiv \alpha_1 \prec \texttt{world}, (\alpha_1 \prec \alpha_i)_{i\in 2..n}, \texttt{this} : c\langle\alpha_{i\in 1..n}\rangle \\ E \vdash c'\langle\sigma\rangle \qquad \mathsf{owner}(c\langle\alpha_{i\in 1..n}\rangle) = \mathsf{owner}(c'\langle\sigma\rangle) \\ \{f_{i\in 1..m}\} \cap \mathrm{dom}(\mathcal{F}_{c'}) = \emptyset \qquad E \vdash t_{j\in 1..m} \qquad E \vdash meth_{k\in 1..p} \\ \forall m \in \mathsf{names}(meth_{k\in 1..u}) \cap \mathrm{dom}(\mathcal{M}\,\mathcal{T}_{c'}) \quad \left\{ \begin{array}{l} \mathcal{M}\,\mathcal{T}_c(m) \equiv t \to t' \\ \mathcal{M}\,\mathcal{T}_{c'}(m) \equiv t'' \to t''' \\ t \equiv \sigma(t'') \quad t' \equiv \sigma(t''') \end{array} \right. \end{array}}{\vdash \texttt{class } c\langle\alpha_{i\in 1..n}\rangle \texttt{ extends } c'\langle\sigma\rangle\{t_j\ f_{j\in 1..m};\ meth_{k\in 1..p}\}}$$

$\boxed{E \vdash P;e}$ Well-typed program

(PROGRAM)

$$\frac{\vdash class_j \ \forall\, class_j \in P \qquad E \vdash e : t}{E \vdash P;e}$$

Figure 8.7: Typing rules for classes and programs in $\mathsf{JO}_?$.

8.2.1 OAD invariant violations, revisited

One can notice that the "plain" subtyping relation from the original calculus JOE_1 [33] operates with types with *all* known owners and ensures that ownership parameters of these types are *equal* modulo subtyping in both the context of the caller and the callee. This property is referred to as *restricted visibility* [36] and states that types of objects assigned to fields, passed as arguments to method calls, returned from field access or method call must have a well-formed ownership structure (i.e., *visible*) in both the context of the callee and the caller. In the presence of a gradual type system, the type checker may reason only about the parts of types it knows statically. In our case, the known parts of types are the defined owners. Let us consider the code fragment below as an example of incomplete reasoning requiring extra dynamic checks:

```
class D<owner> {
  C<?> f // any instance of C is suitable
}
...
D<p> x = new D<p>();
C<q> y = new C<q>();
x.f = y; // invariant violation
```

Assuming that neither the object referred to by x nor its owner p are *within* q, one can see that the last line of the code fragment violates the OAD invariant. However, this case cannot be handled statically via the type system because of the relaxed requirements for the type C<?> of the field f, which would allow an assignment to occur irrespectively of the actual run-time owners. An appropriate *boundary check* should be performed dynamically to ensure that the target object is permitted to access the value assigned to the field.

However, field assignments are not the only place where some sort of dynamic checks might be needed. With gradual types, the presence of $\lesssim$ in the premises of the rules for method calls and returns enables ownership structure of the type of an argument or a returned value to be refined or coarsened correspondingly. When some components of a type are supposed to be refined, a dynamic *type cast* is needed. The code example below provides intuition on the role of gradual types in method parameter and return types and dynamic type casts.

```
class A<owner> {
  C<?> f; // no owner specified for f
  C<owner> specify() {
    // type refinement
    return this.f; // type cast required
  }
}

class B<owner> {
  C<owner> g; // an owner is specified
  void update(C<owner> h) { this.g = h; }
}

// Definition of the class C
A<p> a = new A<p>();
B<p> b = new B<p>();
C<q> c = new C<q>();

a.f = c; // type coarsening, no type cast required
b.update(a.f); // type refinement, a type cast required
b.update(a.specify()); // no type cast required
```

In general, such program places correspond to whenever a value is assigned to a local variable or field or passed to or returned from a method. In the system we consider

only method calls and returns and assignments to fields, since `let`-expressions are not annotated with types and their types are inferred by the compiler.

Essentially, type casts and boundary checks are two orthogonal procedures. The former are standard for gradual type systems: they perform a postponed check that the run-time structure of a datatype corresponds to the programmer's expectations. The latter are particular to systems with ownership types, as they postpone the check that the program does not violate the OAD invariant.

Later, in Section 8.2.4 we describe a *two-pass type-directed* algorithm to sequentially insert both of these types of checks into the program code.

8.2.2 Syntax of $\mathsf{JO}^{+}_{?}$

The syntax is extended for dynamic type casts and boundary checks, inserted by a compiler.

$$\text{Computations} \quad b \quad ::= \quad \ldots \mid \boxed{\langle t \rangle x} \mid \boxed{x.f \leftarrow y}$$

The statement $\langle t \rangle x$ ensures that the run-time type of an object referred to by x *matches* the type t. The statement $x.f \leftarrow y$ performs the check that a field reference from x to y via the field f does not violate the ownership invariant and then performs the field update atomically. Casts and checks are not supposed to be inserted by the programmer. They are inserted instead by the compiler, as described in Section 8.2.4.

8.2.3 Helper relations and program typing in $\mathsf{JO}^{+}_{?}$

If two types are related via $\lesssim$, there is a freedom to choose the run-time semantics of type casts, moving along either $\sim$ or $\leq$ axis. In the original work on gradual types for objects [189], the authors chose to check the *subtyping* at run-time via type casts (i.e., move along the y-axis on the picture from Figure 8.5). More concretely, given $t \lesssim t''$, an intermediate type t', such that $t \sim t'$, is built *statically*. So, only the subsumption $t' \leq t''$ needs to be checked at run-time, and this is implemented via the mechanism of type casts. In contrast, in our case, the definition of $\lesssim$ already gives us an algorithm to compute an "upper-left mediator" (see Figure 8.5).

Following the rule (GRAD-SUB), we compute the type $c'\langle\sigma'\rangle$ that is on the same class-level as the target type $c'\langle\sigma''\rangle$ for the upcast. The following lemma justifies this computation:

Lemma 8.2.1 (Inversion lemma for $\lesssim$)**.** *If $E;B \vdash t \lesssim t''$, then there exists a type t' such that $E;B \vdash t \leq t'$ and $E;B \vdash t' \sim t''$.*

To construct an "upper-left" mediator type t' in the previous lemma, we use an extra helper function $t \uparrow c$ that computes a supertype of the type t at class c.

Definition 8.2.2 ($\uparrow$)**.**

$$\begin{array}{rcl} c\langle\sigma\rangle \uparrow c & \triangleq & c\langle\sigma\rangle \\ c'\langle\sigma\rangle \uparrow c & \triangleq & d\langle\alpha_j \mapsto \sigma(p_j)_{j\in 1..m}\rangle \uparrow c \\ & & \text{where } \texttt{class } c'\langle\alpha_{i\in 1..n}\rangle \texttt{ extends } d\langle p_{j\in 1..m}\rangle \\ & & \text{and } \texttt{class } d\langle\alpha_{j\in 1..m}\rangle \in P. \\ t \uparrow c\langle\_\rangle & \triangleq & t \uparrow c. \end{array}$$

In words, the partially defined function $\uparrow$ pulls up the information from the substitution σ of the initial type $c\langle\sigma\rangle$ until it reaches the desired superclass c. If the class hierarchy `Object` is reached without making a match, the function is undefined. The following lemma states the basic properties of $\uparrow$.

Lemma 8.2.3 (Basic properties of $\uparrow$)**.** *For all E, B, t, t',*

1. $(t \uparrow t) = t$
2. $(E;B \vdash t) \wedge (E;B \vdash t') \wedge (t \uparrow t' \neq \bot) \Rightarrow E;B \vdash t \leq (t \uparrow t')$
3. $E;B \vdash t \lesssim t' \Rightarrow E;B \vdash (t \uparrow t') \sim t'$.

Proof. By induction on the definition of $\uparrow$ □

Our next step is to figure out which particular checks should be performed at run-time. Actually, all relations between *defined* owners (i.e., non-unknown and non-dependent ones) might be already inferred at the type checking stage and, thus, statically checked via the rules for owner ordering.

The relation $E \vdash t \lhd t'$ states that t *satisfies all constraints imposed by the known owners of* t'. It is used to detect where type casts should be inserted.

Definition 8.2.4 (t is more defined than t')**.**

$$\begin{array}{rcl} E \vdash t \lhd t' & \triangleq & E \vdash t \lesssim t' \text{ and } \forall i \;\; q_i \neq ? \vee p_i \neq q_i \\ & & \text{where } (t \uparrow t') = c\langle p_{i\in 1..n}\rangle \text{ and } t' = c\langle q_{i\in 1..n}\rangle \end{array}$$

If for some concrete owner q_i of the right operand, the corresponding owner of the left operand p_i is either unknown or is some dependent owner, the function returns false. One can see that is not the case when $p_i \neq q_i$ for some i and both p_i and q_i are defined. This is filtered out by the clause $t \lesssim t'$. The situation when the target type (on the left-hand side) is more precise in some of its owners than the source type (in the right-hand side) is typical case for type refinement. We use $\lhd$ to detect where type casts should be inserted. Since the consistency on owners is symmetric, the uncertainty can be caused by the lack of information about owners both from the side of a provided and expected type.

If the information about the first owner parameter of type t of some declared field is not known statically, the OAD invariant cannot be guaranteed. In this case a boundary check should be inserted. The predicate **specified**(t) is true iff a type t provides enough static information about its owners to ensure preservation of the OAD invariant.

Definition 8.2.5 (t specifies its owner)**. specified**$(t) \triangleq p_1 \neq ?$, where $t = c\langle p_{i\in 1..n}\rangle$

The type rules for type casts and boundary checks are presented in Figure 8.8. For $\mathsf{JO}^{+}_{?}$ we use different typing relations, namely, $\vdash^{\mathcal{C}}$ and $\vdash^{\mathcal{C}}_{\mathcal{B}}$ corresponding to type casts and boundary checks, respectively. These two relations are similar to $\vdash$ for $\mathsf{JO}_{?}$. The purpose of each of them is to ensure the specific safety conditions after the corresponding stage of the translation.

Informally, $\vdash^{\mathcal{C}}$ guaranties conformance of run-time and expected types; $\vdash^{\mathcal{C}}_{\mathcal{B}}$ augments $\vdash^{\mathcal{C}}$ with the guaranty of preservation of the OAD invariant. Both these are not the case for the original relation ($\vdash$).

One significant difference between the former typing relation $\vdash$ and the new ones is that all the occurrences of $\lesssim$ in the typing of statements are now concentrated on the rule (T-CAST). In the rest of the $\lesssim$-rules are replaced by $\lhd$ (greyed parts). The rule (T-CHECK) ensures type conformance via $\lhd$, but not the preservation of the OAD invariant: this is postponed until run-time. The rule (T-UPD") is specifically targeted to ensure the OAD invariant.

8.2.4 Type-directed program translation

We adopt the idea of Siek and Taha [189] to define a type-directed type cast insertion relation and extend it with the boundary check insertion relation (Figure 8.9, relations $\overset{\mathcal{C}}{\rightsquigarrow}$ and $\overset{\mathcal{B}}{\rightsquigarrow}$, respectively). We distinguish insertions of type casts and boundary checks as two different procedures. First, type casts are inserted into a program whenever additional information about types needs to be regained. Then the boundary check insertion translation works on the program with inserted casts, so each step of the translation eliminates an aspect of uncertainty caused by incomplete type annotations.

$\boxed{E;B \vdash^c b : t}$ Well-typed computation (under $\vdash^c$)

$$\frac{E;B \vdash y : s \quad E;B \vdash t \quad E;B \vdash s \lesssim t}{E;B \vdash^c \langle t \rangle y : t} \text{(T-CAST)}$$

$$\frac{E;B \vdash z : c\langle\sigma\rangle \quad \mathcal{F}_c(f) = t \quad E;B \vdash s \lhd \sigma_z(t) \quad E;B \vdash y : s}{E;B \vdash^c z.f = y : \sigma_z(t)} \text{(T-UPD')}$$

$$\frac{E;B \vdash y : s \quad \mathcal{M}\,\mathcal{T}_c(m) = (y', t \to t') \quad E;B \vdash z : c\langle\sigma\rangle \quad E;B \vdash s \lhd \sigma_z(t) \quad \sigma' \equiv \sigma \uplus \{y' \mapsto y\}}{E;B \vdash^c z.m(y) : \sigma'_z(t')} \text{(T-CALL')}$$

$\boxed{E \vdash^c t'\ m(t\ y)\{e\}}$ Well-typed method (under $\vdash^c$)

$$\frac{E, y : \mathsf{fill}(y,t); B \vdash e : s \quad E;B \vdash s \lhd t'}{E;B \vdash^c t'\ m(t\ y)\ \{e\}} \text{(METHOD')}$$

$\boxed{E;B \vdash^c_{\mathcal{B}} b : t}$ Well-typed computation (under $\vdash^c_{\mathcal{B}}$)

$$\frac{E;B \vdash z : c\langle\sigma\rangle \quad \mathcal{F}_c(f) = t \quad E;B \vdash y : s \quad E;B \vdash s \lhd \sigma_z(t)}{E;B \vdash^c_{\mathcal{B}} z.f \leftarrow y : \sigma_z(t)} \text{(T-CHECK)}$$

$$\frac{E;B \vdash z : c\langle\sigma\rangle \quad \mathcal{F}_c(f) = t \quad E;B \vdash y : s \quad E;B \vdash s \lhd \sigma_z(t) \quad \textbf{specified}(\sigma_z(t))}{E;B \vdash^c_{\mathcal{B}} z.f = y : \sigma_z(t)} \text{(T-UPD'')}$$

Figure 8.8: Selected typing rules of $\vdash^c$ and $\vdash^c_{\mathcal{B}}$. Greyed parts denote differences with the original static semantics as defined by the relation $E;B \vdash b : t$. Omitted rules are identical to those of the relation $\vdash$.

Type cast insertion

Type cast insertion $\overset{c}{\rightsquigarrow}$ is a first stage of the complete gradually-typed program translation. Figure 8.9 provides the definition of selected rules for the cast insertion relation that specifies the translation. It is written $E \vdash e_1 \overset{c}{\rightsquigarrow} e_2 : t$ for expressions and holds if, under the assumptions from E, expression e_1 is translated into expression e_2 and the type of e_1 is inferred as t according to the relation $\vdash$. It is defined in the same way for methods. For the omitted computations (i.e, new instance allocation or field reference) the translation proceeds to the body of the inner expression. For variables the translation is defined as the identity.

The rules for classes and whole programs are straightforward and omitted. Each inserted cast creates a fresh variable and increases the depth of the processed `let`-expression whenever the consistent-subtyping relation is mentioned in the premise of a typing rule. Type cast insertions are type-guided: no cast is inserted if the predicate $\lhd$ holds on the types being compared. For conditional insertions we define the helper function $\mathcal{C}$, which uses non-recursive local decomposition of an expression e via the context G and optionally inserts type-casts:

$$\mathcal{C}_E\langle t_1, t_2\rangle(e) \quad \triangleq \quad \textbf{if } (E \vdash t_1 \lhd t_2) \textbf{ then } e \textbf{ else } (\texttt{let } y' = \langle t_2\rangle y \texttt{ in } G[y']) \\ \textbf{where } y' \text{ is fresh, } e = G[y]$$

$$G \quad ::= \quad [\,] \mid \texttt{let } x = z.m([\,]) \texttt{ in } e \mid \texttt{let } x = (z.f = [\,]) \texttt{ in } e$$

The following lemma holds for the relation $\overset{c}{\rightsquigarrow}$ with respect to the $\vdash^c$-typing.

Lemma 8.2.6 ($\overset{c}{\rightsquigarrow}$ is $\vdash^c$-sound for expressions)**.** *If* $E \vdash e \overset{c}{\rightsquigarrow} e' : t$ *then* $E \vdash^c e' : t$.

Proof. See Section C.1 of Appendix C for a detailed proof (Lemma C.1.1). □

The corollary follows naturally:

Corollary 8.2.7 ($\overset{c}{\rightsquigarrow}$ is $\vdash^c$-sound for methods)**.** *If* $E \vdash t'\ m(t\ y)\ \{e\} \overset{c}{\rightsquigarrow} t'\ m(t\ y)\ \{e'\}$ *then* $E \vdash^c t'\ m(t\ y)\ \{e'\}$

Proof. See Section C.1 of Appendix C for a detailed proof (Corollary C.1.2). □

$\boxed{E \vdash e \overset{\mathcal{C}}{\rightsquigarrow} e' : t}$ Type cast insertion for expressions

$$\text{(C-UPD)}\quad \frac{\begin{array}{c} E \vdash z : c\langle\sigma\rangle \quad \mathcal{F}_c(f) = t \\ E \vdash s \lesssim \sigma_z(t) \quad E \vdash y : s \\ E, x : \mathsf{fill}(x, \sigma_z(t)) \vdash e_1 \overset{\mathcal{C}}{\rightsquigarrow} e_2 : s' \end{array}}{\begin{array}{c} E \vdash \mathtt{let}\ x = (z.f = y)\ \mathtt{in}\ e_1 \overset{\mathcal{C}}{\rightsquigarrow} \\ \mathcal{C}_E\langle s, \sigma_z(t)\rangle(\mathtt{let}\ x = (z.f = y)\ \mathtt{in}\ e_2) : s' \end{array}}$$

$$\text{(C-CALL)}\quad \frac{\begin{array}{c} E \vdash z : c\langle\sigma\rangle \quad \mathcal{M}\,\mathcal{T}_c(m) = (y', t \to t') \quad E \vdash y : s \\ E \vdash s \lesssim \sigma_z(t) \quad \sigma' \equiv \sigma \uplus \{y' \mapsto y\} \\ E, x : \mathsf{fill}(x, \sigma'_z(t')) \vdash e_1 \overset{\mathcal{C}}{\rightsquigarrow} e_2 : s' \end{array}}{\begin{array}{c} E \vdash \mathtt{let}\ x = z.m(y)\ \mathtt{in}\ e_1 \overset{\mathcal{C}}{\rightsquigarrow} \\ \mathcal{C}_E\langle s, \sigma_z(t)\rangle(\mathtt{let}\ x = z.m(y)\ \mathtt{in}\ e_2) : s' \end{array}}$$

$\boxed{E \vdash t'\ m(t\ y)\{e\} \overset{\mathcal{C}}{\rightsquigarrow} t'\ m(t\ y)\{e'\}}$ Type cast insertion for methods

$$\text{(C-METHOD)}\quad \frac{\begin{array}{c} E \vdash e_1 : s \quad E \vdash s \lesssim t' \quad e_2 = F[z] \\ E, y : \mathsf{fill}(y, t) \vdash e_1 \overset{\mathcal{C}}{\rightsquigarrow} e_2 : s \end{array}}{\begin{array}{c} E \vdash t'\ m(t\ y)\{e_1\} \overset{\mathcal{C}}{\rightsquigarrow} \\ t'\ m(t\ y)\{F[\mathcal{C}_E\langle s, t'\rangle(z)]\} \end{array}}$$

$\boxed{E \vdash e \overset{\mathcal{B}}{\rightsquigarrow} e' : t}$ Boundary check insertion for expressions

$$\text{(B-UPD)}\quad \frac{\begin{array}{c} E \vdash z : c\langle\sigma\rangle \quad \mathcal{F}_c(f) = t \quad E \vdash y : s \\ E \vdash s \lhd \sigma_z(t) \\ E, x : \mathsf{fill}(x, \sigma_z(t)) \vdash e_1 \overset{\mathcal{B}}{\rightsquigarrow} e_2 : s' \end{array}}{\begin{array}{c} E \vdash \mathtt{let}\ x = (z.f = y)\ \mathtt{in}\ e_1 \overset{\mathcal{B}}{\rightsquigarrow} \\ \mathtt{let}\ x = \mathcal{B}\langle\sigma_z(t)\rangle(z.f = y)\ \mathtt{in}\ e_2 : s' \end{array}}$$

Figure 8.9: Selected rules of compilation of $\mathsf{JO}_?$ to $\mathsf{JO}^+_?$: cast and check insertion.

Boundary check insertion

Boundary check insertion $\overset{\mathcal{B}}{\rightsquigarrow}$ is the second stage of the whole translation (Figure 8.9). The translation $\overset{\mathcal{B}}{\rightsquigarrow}$ works on top of a $\vdash^{c}$-well-typed program. The only type of the statement that can be affected by $\overset{\mathcal{B}}{\rightsquigarrow}$ is a field update, since it is only one that can possibly break the OAD invariant.

The helper function $\mathcal{B}$ is defined to replace plain assignments with boundary-checked field assignments whenever insufficient type information about primary owners is provided:

$$\mathcal{B}\langle t\rangle(b) \quad \triangleq \quad \textbf{let } (z.f = y) = b \textbf{ in } (\textbf{if specified}(t) \textbf{ then } b \textbf{ else } z.f \leftarrow y)$$

$$F \quad ::= \quad [\,] \mid \texttt{let}\ z = b\ \texttt{in}\ F$$

For the rest of the statements, expressions and methods, $\overset{\mathcal{B}}{\rightsquigarrow}$ is applied recursively to the inner expression's body. For variables the translation is defined as the identity.

Proposition 8.2.8 (Boundary check insertion and $\vdash^{c}$-typing)**.** *$E \vdash e' \overset{\mathcal{B}}{\rightsquigarrow} e'' : s$ iff $E \vdash^{c} e' : s$.*

The typing judgement $\vdash^{c}_{\mathcal{B}}$ is defined to assign types to expressions, statements and methods after boundary check insertion. The selected rules of the judgement are provided in Figure 8.8. The rules for the rest of statements and expressions are similar to those of $\vdash^{c}$. When there is a boundary check translation of the expression e' to e'', the latter is guaranteed to be $\vdash^{c}_{\mathcal{B}}$-typed in the extended language (Lemma 8.2.9).

Lemma 8.2.9 ($\overset{\mathcal{B}}{\rightsquigarrow}$ is $\vdash^{c}_{\mathcal{B}}$-sound for expressions)**.** *If $E \vdash e' \overset{\mathcal{B}}{\rightsquigarrow} e'' : t$ then $E \vdash^{c}_{\mathcal{B}} e'' : t$.*

Proof. See Section C.1 of Appendix C for a detailed proof (Lemma C.1.4). □

The soundness result for methods follows naturally from Lemma 8.2.9.

Corollary 8.2.10 ($\overset{\mathcal{B}}{\rightsquigarrow}$ is $\vdash^{c}_{\mathcal{B}}$-sound for methods)**.** *If $E \vdash t'\ m(t\ y)\ \{e\} \overset{\mathcal{B}}{\rightsquigarrow} t'\ m(t\ y)\ \{e'\}$ then $E \vdash^{c}_{\mathcal{B}} t'\ m(t\ y)\ \{e'\}$*

Proof. See Section C.1 of Appendix C for a detailed proof (Corollary C.1.5). □

Complete program translation

The whole program translation is defined as the composition of cast and boundary check insertions.

Definition 8.2.11. $E \vdash e \rightsquigarrow e'' : t$ iff $E \vdash e \overset{\mathcal{C}}{\rightsquigarrow} e' : t$ and $E \vdash e' \overset{\mathcal{B}}{\rightsquigarrow} e'' : t$ for some $e' \in \mathsf{JO}_?^+$.

The translation relation $E \vdash e_1 \rightsquigarrow e_2 : t$ can be extended to classes and programs in a straightforward fashion. For instance, we denote $\vdash P_1; e_1 \rightsquigarrow P_2; e_2$ if a program $P_2; e_2$ is obtained from $P_1; e_1$ by the compositional type-directed translation.

The following theorem is a direct corollary of Lemmas 8.2.6 and 8.2.9 and states the type soundness of the complete translation of a $\vdash$-well-typed program with respect for $\vdash^{\mathcal{C}}_{\mathcal{B}}$-typing.

Theorem 8.2.12 (Program translation is $\vdash^{\mathcal{C}}_{\mathcal{B}}$-sound.)**.** *$E \vdash e : t$ implies $E \vdash e \rightsquigarrow e' : t$ for some e'. Furthermore, $E \vdash e \rightsquigarrow e' : t$ for some e implies $E \vdash^{\mathcal{C}}_{\mathcal{B}} e' : t$.*

Proof. By the fact that $E \vdash e : t$ implies $E \vdash e \overset{\mathcal{C}}{\rightsquigarrow} e' : t$ for some e' and applying Lemmas 8.2.6 and 8.2.9 subsequently. □

8.3 Operational semantics of $\mathsf{JO}_?^+$

This section provides a definition of dynamic semantics of $\mathsf{JO}_?$. The rules of the small-step operational semantics of $\mathsf{JO}_?$ are presented in Figure 8.10. The semantics is in the form of a small-step CEK-like abstract machine, with a single-threaded store H, binding environment B and explicit continuations K [79]. We have chosen this model since it can be easily extended with new types of computations and expressions. A continuation K is, informally, a serialized next step of computation:

$$\text{Continuations} \quad K \quad ::= \quad \mathbf{mt} \mid \mathbf{call}(x : (t, \sigma), e, B, K) \mid \mathbf{fail}(K)$$

In some sense, the notion of continuations in a Java-like language is similar to the global program stack, which is the basis of another way to formalize the small-step operational semantics for core Java [159].

The *empty* continuation **mt** corresponds to the empty control stack. which is the case at the beginning and at the correct end of program execution. $\mathbf{call}(x : (t, \sigma), e, B, K)$ describes the discipline of popping the stack when a method ends its execution and its caller's local environment B should be restored with a result assigned to a variable x.

$\boxed{\langle H,B,e,K\rangle \Rightarrow \langle H',B',e',K'\rangle}$ Small-step transition relation for $\mathsf{JO}^{+}_{?}$.

$$\frac{B(y)=\iota \quad H(\iota).f=v}{\langle H,B,\texttt{let } x=y.f \texttt{ in } e,K\rangle \Rightarrow \langle H,B[x\mapsto v],e,K\rangle}\ \text{(E-LKP)}$$

$$\frac{B(y)=\iota \quad B(y')=v \quad H(\iota)=o \quad H'=H[\iota\mapsto o[f\mapsto v]]}{\langle H,B,\texttt{let } x=(y.f=y') \texttt{ in } e,K\rangle \Rightarrow \langle H',B[x\mapsto v],e,K\rangle}\ \text{(E-UPD)}$$

$$\frac{\begin{array}{c}B(y)=\iota \quad B(y')=v \quad H(\iota)=\langle c\langle\sigma'\rangle, f\mapsto v_{f\in \mathrm{dom}(\mathcal{F}_c)}\rangle \\ \mathcal{M}_c(m)=(x',e',\{\alpha_i\mapsto p_{i\in 1..n}\}) \\ B'=\{\alpha_i\mapsto\sigma'(p_i)_{i\in 1..n},\texttt{this}\mapsto\iota,x'\mapsto v\}\end{array}}{\langle H,B,\texttt{let } x:(t,\sigma)=y.m(y') \texttt{ in } e,K\rangle \Rightarrow \langle H,B',e',\mathbf{call}(x:(t,\sigma),e,B,K)\rangle}\ \text{(E-CALL)}$$

$$\langle H,B,y,\mathbf{call}(x:(t,\sigma),e,B',K)\rangle \Rightarrow \langle H,B'[x\mapsto B(y)],e,K\rangle\ \text{(E-RETURN)}$$

$$\langle H,B,y,\mathbf{mt}\rangle \Rightarrow \langle H,B,B(y),\mathbf{mt}\rangle\ \text{(E-FINAL)}$$

$$\frac{\iota \text{ is fresh} \quad H'\equiv H\uplus \iota\mapsto\langle c\langle B(p_{i\in 1..n})\rangle,\overline{f\mapsto \texttt{null}}\rangle}{\langle H,B,\texttt{let } x=\texttt{new } c\langle p_{i\in 1..n}\rangle \texttt{ in } e,K\rangle \Rightarrow \langle H',B[x\mapsto\iota],e,K\rangle}\ \text{(E-NEW)}$$

$$\frac{H;B\vdash t\ltimes t' \quad B(y)=\iota \quad H(\iota)=\langle s,\ldots\rangle \quad \widehat{H}\vdash s\lhd t'}{H;B\vdash \mathsf{cast}(t,y)}\ \text{(CAST-CHECK)}$$

$$\frac{B(y)=\texttt{null}\ \vee\ H;B\vdash \mathsf{cast}(t,y) \quad B'=B[x\mapsto B(y)]}{\langle H,B,\texttt{let } x=\langle t\rangle y \texttt{ in } e,K\rangle \Rightarrow \langle H,B',e,K\rangle}\ \text{(E-CAST1)}$$

$$\frac{B(y)\neq\texttt{null} \quad H;B\nvdash \mathsf{cast}(t,y) \quad K\neq\mathbf{fail}(\_) \quad e=(\texttt{let } x=\langle t\rangle y \texttt{ in } e')}{\langle H,B,e,K\rangle \Rightarrow \langle H,B,e,\mathbf{fail}(K)\rangle}\ \text{(E-CAST2)}$$

$$\frac{B(x)=\iota \quad B(y)=\iota' \quad H(\iota')=\langle c\langle k,\ldots\rangle,\ldots\rangle \quad \widehat{H};\emptyset\vdash \iota\prec k}{H;B\vdash \mathsf{boundary}(x,y)}\ \text{(BOUNDARY-CHECK)}$$

$$\frac{(B(y')=\texttt{null}\ \vee\ H;B\vdash \mathsf{boundary}(y,y')) \quad B(y)=\iota \quad B(y')=v \quad H(\iota)=o \quad H'=H[\iota\mapsto o[f\mapsto v]] \quad B'=B[x\mapsto v]}{\langle H,B,\texttt{let } x=(y.f\leftarrow y') \texttt{ in } e,K\rangle \Rightarrow \langle H',B',e,K\rangle}\ \text{(E-BOUNDARY1)}$$

$$\frac{B(y')\neq\texttt{null} \quad K\neq\mathbf{fail}(\_) \quad H;B\nvdash \mathsf{boundary}(y,y') \quad e=(\texttt{let } x=(y.f\leftarrow y') \texttt{ in } e')}{\langle H,B,e,K\rangle \Rightarrow \langle H,B,e,\mathbf{fail}(K)\rangle}\ \text{(E-BOUNDARY2)}$$

Figure 8.10: Small-step operational semantics of $\mathsf{JO}^{+}_{?}$.

The variable x, to which the result of the method will be assigned, is annotated with the declared return type t at the callee's side and the local substitution σ to be applied to t at the caller's side. These annotations can be obtained during the type-checking phase via the rule (T-CALL); they do not affect the dynamic semantics and are used only in the soundness proof. Finally, **fail**(K) denotes the result of failing casts and boundary checks.

To implement dynamic type casts, we first need a bit of machinery to relate *syntactic* types with *dynamic* types extracted from the object heap during program execution. We define a helper relation $H; B \vdash t \ltimes t'$ for *type instantiation* to compute the dynamic type t' corresponding to a static type t in dynamic environments H and B by instantiating owners as follows.

Definition 8.3.1 (Type instantiation)**.**

$$\forall\, i \in 1..n \quad q_i = \begin{cases} k & \text{if } \begin{cases} p_i = x^{c.j} \\ H(B(x)) = \langle t, \ldots \rangle \\ k = \mathsf{owner}_j(t \uparrow c) \end{cases} \quad \text{dependent owner} \\ p_i & \text{if } \mathsf{actual}(p_i) \quad \text{run-time owner} \\ B(p_i) & \text{if } \mathsf{defined}(p_i) \quad \text{formal owner} \\ ? & \text{otherwise} \quad \text{unknown owner} \end{cases}$$

$$H; B \vdash c\langle p_{i\in 1..n}\rangle \ltimes c\langle q_{i\in 1..n}\rangle$$

The statement $\widehat{H} \vdash s \lhd t'$ in the premise of the rule (CAST-CHECK) might seem odd since the check uses not pure subtyping but the "more defined than" relation on types. However, there is nothing wrong since all owners of the left operand s are *known* and to satisfy the relation all the actual owners of its "upper-left" mediator should match actual owners of the type t'. The semantics of type cast only cares about known owners in t'. For the sake of simplicity we do not handle the case $B(x) = \texttt{null}$ in the rules. The rule (BOUNDARY-CHECK) reflects the formulation of the ownership invariant.

The test $\iota \prec k$ in the rule (BOUNDARY-CHECK) can be performed at run-time by checking whether k is ι or some transitive owner of ι—this information is obtained via the flattened heap $\widehat{H}$.

There is some space for design choices when formulating the last assumption of the premise $\widehat{H} \vdash \iota \prec k$. For instance, we could have used the following statement instead:

$$H(\iota) = \langle d\langle p_{i\in 1..m}\rangle, \ldots \rangle \quad k_1 \in p_{i\in 1..m}$$

It is a stronger assumption since in this case an owner k_1 of the object y is required to be one of owner parameters of the object referred to by x. The static version of this

assumption was formulated in the statement of the original OAD invariant by Clarke et al. [36].

Indeed, this premise wold reflect better the typing rule (CLASS) but, in fact, we do not need it to be so strong to preserve the OAD invariant. Surprisingly, ignoring other owner parameters of y, as we did in the rule (BOUNDARY-CHECK), still ensures that the invariant holds! The *local* order of owner parameters with respect to the primary owner imposed by the rule (CLASS) is sufficient to preserve the global ownership structure.

8.4 Type safety

In the concluding section of this chapter, we state the type safety of $\mathsf{JO}_?$ as a corollary of the correctness of the type-guided program translation with respect to program typing and the type safety of the extended language $\mathsf{JO}_?^+$ with type casts and boundary checks. We also show that well-typed programs in $\mathsf{JO}_?^+$ preserve the OAD invariant dynamically. We do *not* formulate a *progress* theorem explicitly [212], focusing mainly on the subject reduction and the OAD invariant preservation. We argue that progress in $\mathsf{JO}_?^+$ is ensured by the type system of the original calculus JOE_1 by Clarke and Drossopoulou [33], since for most of the statements of the typing relation $\vdash^{\mathcal{C}}_{\mathcal{B}}$, excluding those for specific computations of $\mathsf{JO}_?^+$, form a subset of JOE_1's typing relation $\vdash$ (with no effect annotations taken into the account). The proof of progress for the newly introduced computations in $\mathsf{JO}_?^+$, i.e, dynamic type casts and boundary checks, is trivial. A complete formal treatment with the of theorems from this section is available in the Appendix C.

To recall how compilation in $\mathsf{JO}_?$ is related with typing, we start from the following proposition, which is a straightforward corollary of Theorem 8.2.12 and gives one an indication about the programs that can be compiled:

Proposition 8.4.1 (Compilation and gradual typing)**.** $E \vdash P; e \quad \textit{iff} \quad \exists P', e'.\ E \vdash P; e \rightsquigarrow P'; e'$.

That is, well-formedness of a program guarantees its compilation.

To prove the preservation of the OAD invariant, Clarke and Drossopoulou introduce the notion of a *well-formed* heap H with respect to the type annotations provided and a well-formed context-binding pair $E; B$ [33, Section 6.3]. Unfortunately, this approach does not work well in the presence of gradual types, since unknown owners prevent us from reasoning about the precise correspondence between heap objects and syntactic owners.

The preservation of the OAD invariant in our calculus relies instead on three facts:

1. The initial configuration of any program obeys the OAD invariant;
2. The subject reduction theorem guarantees well-formedness for a subsequent configuration;
3. Making a step from any well-formed configuration obeying the OAD invariant, preserves the invariant.

In the remainder of this section we formalize these statements.

One can notice that in the typing rules for $\mathsf{JO}_?^+$, the consistent-subtyping relation is present only in the rule (T-CAST) for dynamic type casts, i.e., those inserted by the compiler, as was described in Section 8.2. This phenomenon is typical for gradual type systems: all relaxed assumptions about types are concentrated in rules for dynamic check statements. This design of a type system results in the following interesting property that a fully-annotated well-typed program in $\mathsf{JO}_?$ will not result in the **fail** continuation. At the same time, a program with allnot-necessary annotations omitted may still end with **fail** continuation if one tries to assign some object o to inappropriate field, which would cause to o to escape its owner's context.

8.4.1 Typing dynamic environments

The operational formalism we use is an abstract CEK-machine with a heap, so we need to separate environments to provide typing for heap objects and references in stack frames. We define heap and stack environments as follows:[2]

$$\begin{array}{lrcl} \text{Heap environments} & \mathcal{E} & ::= & \emptyset \mid \mathcal{E}, \iota : c\langle k_{i\in 1..n}\rangle \mid \mathcal{E}, \iota \prec k \\ \text{Stack environments} & \overline{E} & ::= & \textbf{nil} \mid E :: \overline{E} \end{array}$$

In the remainder of this section, we assume that static typing environments E defined in Section 8.1 contain only term and owner variables in their domain, but not heap locations.

Figure 8.11 describes the relation of well-formed triples ($\mathcal{E}, E; B \vdash \diamond$), equivalences between static and dynamic owners ($\mathcal{E}, E; B \vdash p = p'$) and well-formed continuations ($\mathcal{E}; \overline{E}; B \Vdash \langle e, K\rangle$). Although the rule (BINDING-VALUE) for well-formed environment-binding pairs might look a bit complicated, it is essential to handle dependent owners correctly an run-time. In words, it ensures that the actual run-time type of the value v is a proper subtype of the static type $c\langle\sigma\rangle$ (modulo equivalences)

[2] Even though the judgments of the form $\iota \prec k$ in heap environments can be derived from the heap entries $\iota : c\langle k_{i\in 1..n}\rangle$, we made them explicit in the environment in order to simplify the proofs.

with dependent owners $z^{c.j}$ replaced by runtime owners of t, lifted via the helper function $t \uparrow c$ to the level of a class c.

Figure 8.12 defines well-formed run-time objects ($\mathcal{E} \vdash \iota \mapsto o : t$), heaps ($\mathcal{E} \vdash H$) and, eventually, well-formed program states ($\mathcal{E}, \overline{E} \Vdash \langle H, B, e, K \rangle$). The last clause $\widehat{H} \Rightarrow \mathcal{E}$ in the premise of the rule (HEAP) is the key ingredient to define *correct* run-time heaps. It states that the environment $\mathcal{E}$ provides *no more* information than can be obtained from the flattened heap via the standard rules.

Definition 8.4.2 (Heap entailment)**.** $\mathcal{E} \Rightarrow \mathcal{E}'$ iff $\mathcal{E}, \emptyset \vdash \mathcal{H}$ for all statements $\mathcal{H} \in \mathcal{E}'$

Note that *well-formedness* of an environment $\mathcal{E}$ is an important requirement. Otherwise, we would end up with "well-formed" heaps that after flattening would exhibit cyclic nesting chains (i.e., $\iota \prec \ldots \iota' \ldots \prec \iota$ for some $\iota \neq \iota'$) thereby depriving the definition of the owners-as-dominators invariant any sense.

A stack environment is *well-formed* if all its constituents are well-formed. The definition of a well-formed run-time state ($\mathcal{E}, \overline{E} \Vdash \langle H, B, e, K \rangle$, Figure 8.12) assumes the expression e to be well-typed ($\mathcal{E}, E_0; B \vdash^{c}_{\mathcal{B}} e : t$) and that environments $\mathcal{E}$ and $\overline{E}$ are well-formed. The last clause ensures, in particular, that the heap H has no ownership cycles ($\mathcal{E} \vdash H$), otherwise the well-formedness of the typing environment $\mathcal{E}$ would be violated.

The following lemma describes the valid shape of an initial program state.

Lemma 8.4.3 (Initial state typing)**.** $\mathcal{E}, E; B \vdash^{c}_{\mathcal{B}} e : t$ *iff* $\mathcal{E}, (E :: \mathbf{nil}) \Vdash \langle H, B, e, \mathbf{mt} \rangle$ *for some initial heap* H *such that* $\mathcal{E} \vdash H$.

Proof. By the rule (T-STATE), Figure 8.12. □

8.4.2 Subject reduction

In this section we establish a subject reduction result for program states in $\mathsf{JO}^{+}_{?}$. That is, making a step from a well-formed state yields another state that is also well-formed.

Definition 8.4.4 (Heap environment extension)**.** An environment $\mathcal{E}'$ is an extension of $\mathcal{E}$ (written $\mathcal{E}' \gg \mathcal{E}$) if and only if $\mathcal{E} \subseteq \mathcal{E}'$.

Definition 8.4.5 (Stack environment evolution)**.** We say that a stack environment $\overline{E}$ transforms to a stack environment $\overline{E}'$ (written $\overline{E} \twoheadrightarrow \overline{E}'$) if one of the following holds:

- $\overline{E}' = E' :: \overline{E}$ for some E' (method call);
- $\overline{E}' = (E_0, x : t) :: \mathsf{tail}(\overline{E})$ for some t and $x \notin \mathrm{dom}(E_0)$ (variable assignment);

$\boxed{E;B \vdash \diamond}$ Well-formed environment-binding pair

$$\frac{\mathcal{E},E \text{ is well-formed}}{\mathcal{E},E;\emptyset \vdash \diamond}\ (\text{BINDING-EMPTY}) \qquad \frac{\alpha \prec p \in E \quad \mathcal{E},E;B \vdash k \prec p \quad \alpha \notin \mathrm{dom}(B)}{\mathcal{E},E;B,\alpha = k \vdash \diamond}\ (\text{BINDING-OWNER})$$

$$\frac{\mathcal{E},E;B \vdash v : t \quad z : t' \in E \quad t' = c\langle\sigma\rangle \quad z \notin \mathrm{dom}(B) \quad \mathcal{E},E;B \vdash t \leq c\langle\sigma \uplus \{z^{c.j} \mapsto \mathsf{owner}_j(t \uparrow c)\}\rangle}{\mathcal{E},E;B,z = v \vdash \diamond}\ (\text{BINDING-VALUE})$$

$\boxed{E;B \vdash p = p'}$ Owners equality

$$\frac{\mathcal{E},E;B \vdash \diamond \quad \alpha = k \in B}{\mathcal{E},E;B \vdash \alpha = k}\ (\text{IN-BIND1}) \qquad \frac{\mathcal{E},E;B \vdash \diamond \quad z = \iota \in B}{\mathcal{E},E;B \vdash z = \iota}\ (\text{IN-BIND2})$$

$$\frac{\mathcal{E},E;B \vdash \diamond \quad \mathcal{E},E;B \vdash z = v \quad \mathcal{E},E;B \vdash z : c\langle\sigma\rangle \quad \mathcal{E},E;B \vdash v : t}{\mathcal{E},E;B \vdash z^{c.i} = \mathsf{owner}_i(t \uparrow c)}\ (\text{IN-BIND3})$$

$\boxed{\mathcal{E};\overline{E};B \Vdash \langle e,K\rangle}$ Well-formed continuation

$$\frac{\mathcal{E},E_0;B \vdash^{C}_{\mathcal{B}} e : s \quad \mathcal{E},E_0;B \vdash s \lhd t \quad \forall p \in \mathrm{dom}(\sigma) \quad (\mathcal{E},E_0,B \vdash p = p') \Leftrightarrow (\mathcal{E},E_1,B' \vdash \sigma(p) = p') \quad \mathcal{E},E_1,(x : \mathsf{fill}(x,\sigma(t))),\overline{E};B' \Vdash \langle e',K\rangle}{\mathcal{E};E_0 :: E_1 :: \overline{E};B \Vdash \langle e,\mathbf{call}(x : (t,\sigma),e',B',K)\rangle}\ (\text{TC-CALL})$$

$$\frac{\mathcal{E},E;B \vdash^{C}_{\mathcal{B}} e : t}{\mathcal{E};E :: \mathbf{nil};B \Vdash \langle e,\mathbf{mt}\rangle}\ (\text{TC-MT}) \qquad \frac{\mathcal{E};\overline{E};B \Vdash \langle e,K\rangle}{\mathcal{E};\overline{E};B \Vdash \langle e,\mathbf{fail}(K)\rangle}\ (\text{TC-FAIL})$$

Figure 8.11: Well-formed bindings and continuations, owners equality.

$\boxed{\mathcal{E} \vdash \iota \mapsto o : t}$ Well-formed object

$$\frac{\begin{array}{c}(\text{HEAP-OBJECT}) \\ o \equiv \langle c\langle\sigma\rangle, \overline{f \mapsto v}\rangle \\ \mathcal{E};\emptyset \vdash c\langle\sigma\rangle \quad \mathcal{E};\emptyset \vdash \iota \prec \mathsf{owner}(c\langle\sigma\rangle) \\ \mathcal{E};\emptyset \vdash v_f : s \quad \mathcal{E} \vdash s \lhd \sigma_\iota(\mathcal{F}_c(f)) \quad \forall f \in \mathrm{dom}(\mathcal{F}_c)\end{array}}{\mathcal{E} \vdash \iota \mapsto o : c\langle\sigma\rangle}$$

$\boxed{\mathcal{E} \vdash H}$ Well-formed heap

$$\frac{\begin{array}{c}(\text{HEAP}) \\ \mathcal{E} \text{ is well-formed} \\ \iota : t \in \mathcal{E} \quad \mathcal{E} \vdash \iota \mapsto o : t \\ \forall \iota \mapsto o \in H \quad \widehat{H} \Rightarrow \mathcal{E}\end{array}}{\mathcal{E} \vdash H}$$

$\boxed{\mathcal{E}, \overline{E} \Vdash \langle H, B, e, K\rangle}$ Well-formed state

$$\frac{\begin{array}{c}(\text{T-STATE}) \\ \mathcal{E} \vdash H \quad \mathcal{E};\overline{E};B \Vdash \langle e, K\rangle\end{array}}{\mathcal{E};\overline{E} \Vdash \langle H, B, e, K\rangle}$$

Figure 8.12: Well-formed objects, heaps and states.

- $\overline{E}' = (E_1, x : t) :: \mathsf{tail}(\mathsf{tail}(\overline{E}))$ for some t and $x \notin \mathrm{dom}(E_1)$ (method return).

The functions head and tail are defined for stack environments as standard ones for lists. We use the notation $E_0 = \mathsf{head}(\overline{E})$, $E_1 = \mathsf{head}(\mathsf{tail}(\overline{E}))$ etc.

Theorem 8.4.6 states the subject reduction invariant. The type preservation result is specific for $\mathsf{JO}^+_?$ because of the premise containing $\lhd$ in the rule (HEAP-OBJECT). This is the way the aliasing is controlled in $\mathsf{JO}^+_?$: an object aliased by a field conforms to the signature of the field it is referred to by (in terms of concrete owners). It is worth noticing that this is not always the case in $\mathsf{JO}_?$.

Theorem 8.4.6 (Subject reduction in $\mathsf{JO}^+_?$)**.** *If $e \in$ **Expr** in $\mathsf{JO}^+_?$, $\mathcal{S} = \langle H, B, e, K\rangle$, $\mathcal{E}, \overline{E} \Vdash \mathcal{S}$ for some well-formed $\mathcal{E}, \overline{E}$ and $\mathcal{S} \Rightarrow \mathcal{S}'$, then $\mathcal{E}', \overline{E}' \Vdash \mathcal{S}'$ for some well-formed $\mathcal{E}', \overline{E}'$ such that $\mathcal{E}' \gg \mathcal{E}$ and $\overline{E} \twoheadrightarrow \overline{E}'$.*

Proof. See Section C.2 of Appendix C for a detailed proof (Theorem C.2.16). □

8.4.3 OAD invariant preservation

Theorem 8.4.7 ensures that for all well-formed states, if it is possible to make a next step in the operational semantics, then the OAD invariant is preserved for the heap component of the resulting state.

Theorem 8.4.7 (OAD preservation in $\mathsf{JO}_?^+$)**.** *If* $e \in$ **Expr** *in* $\mathsf{JO}_?^+$, $\mathcal{S} = \langle H, B, e, K\rangle$, $\mathcal{E}; \overline{E} \Vdash \mathcal{S}$, $\mathsf{OAD}(H)$ *and* $\mathcal{S} \Rightarrow \mathcal{S}'$ *for some* $\mathcal{S}' = \langle H', \_, \_, \_\rangle$ *then* $\mathsf{OAD}(H')$.

Proof. See Section C.2 of Appendix C for a detailed proof (Theorem C.2.17). □

8.4.4 Static type safety of $\mathsf{JO}_?$

In order to characterize static type safety of $\mathsf{JO}_?^+$, we need to correctly handle all possible execution scenarios. We define the predicate NPE for *null-pointer error* on states to consider the result of dereferencing fields pointing to `null`.

Definition 8.4.8 (Null-pointer error states)**.** The state $\mathcal{S} = \langle H, B, e, k\rangle$ is *stuck* due to *dereferencing a null-pointer* ($\mathsf{NPE}(\mathcal{S})$) iff $e = D[y]$ for some y and $B(y) = \texttt{null}$, where null evaluation contexts D [155] are defined below:

$$D \quad ::= \quad \texttt{let}\ x = [\,].m(y')\ \texttt{in}\ e \ \mid\ \texttt{let}\ x = ([\,].f = y')\ \texttt{in}\ e \ \mid\ \texttt{let}\ x = [\,].f\ \texttt{in}\ e$$

The NPE-states are terminal for execution traces in the semantics of $\mathsf{JO}_?$/ $\mathsf{JO}_?^+$, since there are no transition rules for them. They could be handled statically by using another pluggable type system, such as one described in [77]. We avoid addressing `NonNull`-annotations and corresponding static safety results in this work.

Definition 8.4.9 (Initial state)**.** Assume $P; e$ to be a program in $\mathsf{JO}_?^+$, and that expression e has no free variables (except `this`), $H = \{\texttt{world} \mapsto \bullet\}$, $B = \{\texttt{this} \mapsto \texttt{world}\}$ is an *initial binding environment*. Then the *initial configuration* of $P; e$ is $\mathbf{init}(e) = \langle H, B, e, \mathbf{mt}\rangle$.

Following [36], we introduce a singleton class `World` with no owner parameters to represent the unique object corresponding to the owner of `world`-annotated instances, and for the completeness we need to provide its type. Taking $\mathcal{E} = \{\texttt{world} : \texttt{World}\}$ and $\overline{E} = \{\texttt{this} : \texttt{World}\} :: \mathbf{nil}$, we obtain $\emptyset \vdash^c_{\mathcal{B}} P; e \Rightarrow \mathcal{E}, \overline{E} \Vdash \mathbf{init}(e)$ by Lemma 8.4.3. Theorem 8.4.10 ends our chain of safety statements.

Given a well-typed program in JOE_1, it can be translated to a $\vdash^c_{\mathcal{B}}$-typed program in $\mathsf{JO}_?^+$. The latter is then embedded into an initial state $\mathcal{S}$ via Definition 8.4.9, and $\mathcal{S}$ is well-formed by construction. Finally, a sequence of steps from the well-formed

initial state leads to a well-formed state (Theorem 8.4.6) again, and each step preserves the OAD invariant (Theorem 8.4.7). The following result ensures that every well-compiled program will either not terminate, or execute to the end state, or will fail because of dynamic type cast or boundary check failure.

Theorem 8.4.10 (Static type safety of $\mathsf{JO}_?$)**.** *If* $\vdash P; e \rightsquigarrow P'; e'$ *and* $\mathbf{init}(e') \Rightarrow^* \mathcal{S}$*, then one of the following statements holds:*

(a) $\mathcal{S} = \langle H, B, v, \mathbf{mt} \rangle$ *for some* H, B *and* v *(final state);*
(b) $\mathsf{NPE}(\mathcal{S})$ *(null-pointer error);*
(c) $\exists \mathcal{S}' : \mathcal{S} \Rightarrow \mathcal{S}'$ *(progress);*
(d) $\mathcal{S} = \langle H, B, b, \mathbf{fail}(K) \rangle$*, where* $b = \langle t \rangle y$ *or* $b = z.f \leftarrow y$ *for some* H, B, t, y, z, f *and* K *(*OAD *violation attempt).*

Proof. Follows immediately from Theorems 8.2.12, 8.4.6 and 8.4.7 and well-typedness of the initial program state. □

Combined Theorems 8.2.12, 8.4.7 and 8.4.10 state that the provided gradual type system ensures that

(a) During the execution of a partially-annotated compiled program *no* ownership invariant will be violated, but a type cast of boundary check failure may occur due to the dynamic invariant violation.

(b) A fully-annotated well-typed program will be executed until the final or null-pointer error state with *no* ownership invariant violation, or will not terminate.

Chapter 9

Implementation and Evaluation

> The gap between theory and practice is
> not as wide in theory as it is in practice.
>
> *Author unknown*

In this chapter, we present a prototype implementation of the front-end of a translating compiler for gradual ownership types and discuss the main challenges that occurred and design choices made during the implementation of the compiler. We also provide a short report about the migration of several classes from the Java Collection Library to support ownership types and the owners-as-dominators invariant.

9.1 Implementation

A prototype compiler for Gradual Ownership Types has been implemented using the JastAdd framework as an extension of the JastAddJ compiler for Java [74]. Thanks to the aspect-based model of JastAdd, no original implementation code needed to be changed. The extension is about 2,600 lines of code, not including tests, blank lines and comments. A prototype is available on GitHub:

`http://github.com/ilyasergey/Gradual-Ownership`

Although generics were introduced in Java 5, we have chosen Java 1.4 as the host language for the sake of simplicity. Parametric polymorphism is an orthogonal feature to ownership parametrization, although recent research has shown that they can be unified [170].

The type analysis and type-directed translation are implemented as attributes in the reference attribute grammar framework [74]. The type analysis is built on top of the standard Java type-checking algorithm, which is augmented to handle ownership-parametrized types. The compiler uses several default conventions as well as *manifest ownership* [37] to seamlessly embed the *raw* Java code into an ownership-aware environment. Similarly to the $\mathsf{JO}_?$ calculus, class and interface ownership parameters can be used to indicate ownership statically, as well as method final parameters, local variables and class instance fields. To be parametrized by some owners, a class or an interface requires *all* its super classes and the interfaces it implements to carry ownership parameters. This is to ensure that no casts of ownership-parametrized types to *raw* types are allowed, since it could lead to a violation of the OAD invariant [170]. The only exception to this rule is the handling of `Object` class. We assume that two `Object` classes exist: one is ownership-parametrized and the other is owned by **`world`** and considered as a special case of the first. If a class does not extend the standard, non-parametrized version of `Object` *explicitly*, it is allowed to declare ownership parameters. Classes that inherit from parametrized classes or interfaces but do not declare ownership parameters are implicitly assumed to be owned by **`world`**, which is made the owner of their supertypes.

The developed compiler provides hints for easily migrating to ownership types by emitting static error messages and warnings. A static error message is emitted whenever necessary annotations are omitted. A warning message is displayed whenever dynamic casts or boundary checks need to be inserted.

9.1.1 Program transformation

We discuss the implementation of the particular parts of the translation using a familiar example, depicted in Figure 9.1.

The type-directed translation is implemented as a source-to-source transformation by erasing ownership types, augmenting classes with fields for owner parameters and inserting run-time checks into the code of expressions. The compiler may also need to modify code that interacts with owner-parametrized classes, i.e, some client applications might need to be recompiled. Therefore, the interoperability of our approach with compiled libraries is still an issue to be addressed in the future work.

The result of the type-directed translation with some implementation details omitted is shown in Figure 9.2.

```
class E<P> {
  D d = new D<P>();
}

class D<owner> {
  E<owner> e;
  void use(D<owner> arg) {}
  void exploit(E<owner> arg) { this.e = arg; }

  void test() {
    final E e = new E<this>();
    final D d = e.d; // boundary check required
    d.use(d);
    d.exploit(e);    // dynamic cast required for e
  }
}
```

Figure 9.1: A partially-annotated program with gradual ownership types.

9.1.2 Implementing ownership parameters

To enable dynamic checks of the ownership structure, auxiliary final private fields to store the ownership parameters of a class are generated during the translation. These fields are initialized immediately in the primary constructor of the class. In the case of a call to a super-class constructor, the initializers for ownership fields are inserted right after the **`super`**`()` statement. The compiler will also change the signatures of all constructors of the processed class: their parameter lists will be extended in front with additional parameters for owners.

The generated private owner fields and utility methods are placed at the beginning of each class' body. After the translation ownership parametrization is erased. For instance, the classes `E` and `D` in Figure 9.2 now have private fields `P` and `owners` to represent owners, respectively. We do not alpha-rename ownership parameters in order to avoid name conflicts with actual fields, but provide necessary checks for name clashes instead. The generated method `__owner()` in each case is used to retrieve the primary owner of a class.

9.1.3 Implementing dependent owners

Instead of transforming Java programs into ANF, the compiler expresses dependent owners in terms of *source code locations* (i.e., encoded positions in a source file) corresponding to the expression that computes the owned object. For example, consider the code in Figure 9.1. The expression `e.d` in the statement **`final`** `D d = e.d` is

```
class E {
  // Generated owner fields and utility methods
  final private Object P;
  public Object __owner(){ return E.this.P; }
  public E __castAs_E(Object P) {...}
  public D __safeUpdate_d(D o) {...}

  // Inner utility method for boundary checks
  private boolean __insideOwnerOf(Object o) throws ... {...}

  // Transformed class body
  E(Object P) {
    super(); this.P = P;
  }

  D d = new D(E.this.P);
}

class D {
  // Generated owner fields and utility methods
  final private Object owner;
  public Object __owner() { return D.this.owner; }
  public D __castAs_D(Object owner) {...}

  // Inner utility methods for null-pointer safety
  private static D D_put(D _arg, Map map, String loc) {...}
  private static E E_cast(E _arg, Object P) {...}

  // Transformed class body
  D(Object owner) {
    super(); this.owner = owner;
  }

  E e;
  void use(D arg) {  }
  void exploit(E arg) { this.e = arg; }
  void test() {
    final Map _depMap = new HashMap();
    final E e = new E(D.this);
    final D d = D_put(e.d, _depMap, "11_19");
    d.use(d);
    d.exploit(E_cast(e, _depMap.get("dep_11_19_D_owner")));
  }
}
```

Figure 9.2: The result of translation of the code from Figure 9.1. Fully-qualified class names and bodies of the generated utility methods are omitted for the sake of clarity.

typed with a dependent owner due to the absence of ownership parametrization of a field `d` in the class `E`. The "label" assigned to this expression is retrieved from its position as `11_19` (Figure 9.2).

The utility static method for maintaining a local map of dependent owners, `D_put()`, helps to deal with owners of anonymous expressions, which can be used as dependent owners. The operation `D_put()` in Figure 9.2 binds the dependent owner with the position `loc` in the locally-allocated map, represented by a passed argument `map` with the owner of the receiver instance object of type `D` and returns the object of type `D` unchanged. This method is implemented as a local static method in order to deal with cases where the expression with producing a dependent owner is `null`, so the recording of a dependent owner will not lead to a null-pointer exception. In the case of non-`null` object, the corresponding dependent owner can be later retrieved using the standard `Map`'s method `get()`.

We rely on the computational semantics of Java, so no dependent owner can be requested before it has been initialized. Moreover, if the expression at some location has been recalculated, its dependent owners will be also recalculated.[1]

Any expression in the program can give rise to dependent owners, which potentially can be used in further checks. To avoid having to manage all possible source locations, the compiler runs a simple static analysis to determine which dependent owners might be used in the current context. If no dependent owners are involved, no local map of dependent owners will be generated. This is the case, for instance, in methods `use()` and `exploit()` of the class `D`. The class `E` does not have a method similar to `D_put()`, because there are no methods operating with dependent owners in `E`.

9.1.4 Implementing casts and boundary checks

The cast operation is class-specific and is implemented via methods like `__castAs_E()` and `__castAs_D()` in Figure 9.2. These methods take the expected static owners and perform a dynamic check of the ownership structure of the object using its owner fields. Local static helper methods such as `E_cast()` are generated in order to provide succeeding casts for the `null` reference. Figure 9.3 provides details of the generated implementation of a method `__castAs_E()` of the class `E`.

A run-time exception containing debugging information about owners will be thrown if a cast fails, otherwise the same object will be returned. The class-specific cast operation implemented this way is polymorphic, however, because of the lack of the parametric polymorphism and covariant method result overriding in Java 1.4, the

[1] Note that mutable variables cannot be used as qualifiers for dependent owners, so the result of the translation refers to the location of the last variable reference instead. In contrast, for final variables owners are inferred.

```
public E __castAs_E(Object P) {
  final Object _P = E.this.P;
  if (_P.equals("World") && P.equals("World")) {/* it's ok */}
  else if (P.equals("UnknownOwner")) {/* it's ok */}
  else if (P != _P) {
    throw new Error("Owner parameter equality error: " + debugInfo(P));
  }
  return this;
}
```

Figure 9.3: Generated machinery for the type cast in the class `E` from Figure 9.2.

utility cast-methods are implemented via massive code duplication in the inheriting classes.

The machinery for boundary checks is implemented in a similar way. Utility methods are generated only for updates of the fields that do not specify their primary owners. An example of a boundary check is the method `__safeUpdate_d()` of the class `E` in Figure 9.2. The details of the generated implementation is shown in Figure 9.4. In order to check the transitive ownership when updating a field, we make use of Java's reflection mechanism to retrieve primary owners by using the method `__owner()` and calling the owner's method `__insideOwnerOf()`.

There are no boundary checks methods in class `D` as it specifies the primary owner of its field `e`. Ownership-parametrized fields with unknown owners may occur even in **`world`**-owned classes. Therefore, the translation will affect them as well.

9.1.5 Supporting inner classes via manifest ownership

In Java a non-static inner class is nested in the body of another class and contains an implicit reference to its enclosing instance (the *outer instance*). Although private inner classes enable some instance-specific subroutines to be encapsulated, an instance of such a class can be leaked and referred to through a field by another object *outside* of its outer instance, which, again, may break the desired invariant. There are multiple suggestions on the problem of interoperation of inner classes and different ownership policies. For example, a solution proposed initially by Clarke [37] and elaborated later by Boyapati et al. [23] allows inner classes to violate the owners-as-dominators invariant. The proposed variant of ownership types allows inner class objects to have privileged access to the representations of the corresponding outer class objects, which, in principle, makes it possible for an inner class to escape its outer instance's scope by being assigned to an external reference. This leads to a weaker statement of

```
public D __safeUpdate_d(D o) {
  if (o == null) return o;
  boolean result;
  try { result = this.__insideOwnerOf(o); }
  catch (Exception e) {
    throw new Error("Unexpected error in method __insideOwnerOf()");
  }
  if (result) {
    return (this.d = o); // perform update and return the result
  } else {
    throw new Error("Ownership invariant violation: " + debugInfo(o));
  }
}

// An utility method to check the nesting relation ≺
private boolean __insideOwnerOf(Object o) throws ... {
  java.lang.reflect.Method m;
  try {m = o.getClass().getMethod("__owner", new Class[0]);}
  catch (NoSuchMethodException e) {
    /* o's owner is world (implicitly) */
    return true;
  }

  Object oOwner = m.invoke(o, new Object[0]);   // get o's owner
  /* Reflexivity */
  if (oOwner == this) return true;
  /* o's owner is world (explicitly) */
  if (oOwner.equals("World")) return true;
  /* my owner is o's owner */
  Object myOwner = this.__owner();
  if (myOwner == oOwner || myOwner == o) return true;

  java.lang.reflect.Method m1;
  try {m1 = myOwner.getClass().getMethod("__owner", new Class[0]);}
  catch (NoSuchMethodException e) {
    /* o's owner is not world, but mine is world */
    return false;
  }

  /* Transitivity */
  java.lang.reflect.Method insideOwner;
  try {
    insideOwner = myOwner.getClass().getMethod("__insideOwnerOf", ...);
  } catch (NoSuchMethodException e) { return false; }
  Object[] ownerArgs = new Object[]{oOwner};
  if (insideOwner != null) {
     // transitivity
    return ((Boolean)insideOwner.invoke(myOwner, ownerArgs));
  }
  return false;   // failed boundary check
}
```

Figure 9.4: Generated machinery for the boundary checks in the class `E` from Figure 9.2.

the ownership invariant, so the *encapsulation theorem* by Boyapati et al. becomes: *x can access an object owned by o only if:*

1. $x \prec o$, *or*
2. *x is an inner class object of o*.

In contrast with this proposal, we are interested in keeping the traditional ownership invariant (see Definition 8.1.2) for inner classes. Considering owner parameters as permissions to refer to owner objects, one can notice that in instance methods and fields of a class, the following permissions are accessible: **this**, **world** and ownership parameters. All of them correspond to the *immutable* references within the class body. What one expect to have is a way to pass these permission to the body of an inner class.

One possible solution to adapt the ownership type system to handle inner classes would be to implicitly assume that inner classes are located within the same encapsulation boundaries as their outer class. This solution, though, would impose too much of restriction on inner class instances as they would be implicitly ownership-parametrized and restricted to some particular encapsulation boundary, which might have not been a programmer's intention. Instead, we make the outer instance's ownership parameters legal in the scope of an inner class if the programmer declares them in the header of an inner class as owner arguments *explicitly*, i.e., by a sort of closure-conversion. This is the way to prevent an inner class carrying no ownership parameters from wrapping an object, owned by its outer instance, and carrying it away as in the following code:

```
class G<owner> {
  G<this> myOuterField = new G<this>();

  class F {
    G<this> myInnerField = myOuterField;
  }
  /* G.this-owned reference escapes G's encapsulation boundary
     through an inner class instance. */
  F getF() { return new F(); }
}
```

However, most of the time one does not intend an inner class to be parametrized, since it is often intended to be used only within its outer class' body and, thus, deal with only one parametrization, imposed by its outer instance ownership structure. To solve this design problem, we employ *manifest ownership*, a mechanism allowing one to declare owned classes without explicit owner type parameters [37, 170]. A manifest class does not have an explicit owner parameter, rather the class's owners are fixed, so

```
interface Iter<P, Q> {}

class D<owner> {
  class MyItr implements Iter <D.this, D.this.owner> {
    D<D.this.owner> myD = new D<D.this.owner>();
  }
  MyItr getItr() { return  new MyItr(); }
  void main() {
    final D d = new D<this>();
    Iter<d, this> iter = d.getItr();  // ok
    D<this> d1 = iter.myD;            // ok
  }
}
```

Figure 9.5: The definition of an inner class employing manifest ownership (greyed). The owner `D.this` refers to an outer instance of the inner class `MyItr`. The owner `D.this.owner` refers to an outer instance's ownership parameter `owner`.

all the objects of the class have the same owners. In the case of inner classes, the outer instance's parameters can be used to declare the inner class' manifest ownership.

In the current implementation the only way to supply a manifest parametrization for an inner class is to make it extend an owner-parametrized class (or implement an owner-parametrized interface). Figure 9.5 provides an example of employing the manifest ownership for inner classes. We are using qualified names to refer outer instance's **this** and owner parameters. The `MyItr` class is implicitly parametrized by its outer instance and outer instance's owner, thanks to its superclass parametrization (greyed in the code fragment above). The result of this parametrization is illustrated by the body of the method `main()` in Figure 9.5. Thanks to dynamic aliasing and manifest ownership of the class `MyItr`, the type `Iter<d, this>` is a valid supertype of the type `d.MyItr` in a context where `d` is an immutable variable referring to an object of type `D<this>`. Therefore, the assignment `Iter<d, this> iter = d.getItr()` is well-typed.

9.1.6 Gradual ownership types and inheritance

In the current implementation, we provide a syntactic sugar for subclassing, assuming non-parametrized subclasses of parametrized classes to be owned by **world**, i.e., implementing a form of manifest ownership as follows:

```
class S<owner, outer> { ... }

/*
  Implicitly:
  class T extends S<world, world> { ... }
*/
class T extends S { ... }
```

The same default convention is provided for extending interfaces.

Following the *Liskov Substitution Principle*, we implement the following policy with respect to the return types overridden methods: a type with known owners can be used as a substitute for a type with corresponding unknown owners, but not vice versa. This is a slightly weaker policy comparing with the one described by the rule (CLASS), which requires equality of ownership structures in methods' return types (modulo class-specific substitutions). In fact, the covariant subtyping of method return types with respect to the order $\lhd$ is still sound with respect to OAD preservation, so we allow it.

Let us consider the following example of valid method overriding:

```
// ◁-covariant method return type overriding
class A<aOwner> {
  A<?> foo(A<?> param) { ... }
}
class B<bOwner> extends A<bOwner> {
  A<bOwner> foo(A<?> param) { ... } // ok
}
class C<cOwner> extends B<cOwner> {
  A foo(A<?> param) { ... } // return type is A<cOwner>
}
```

Method `foo()` in the class `B` is overridden by providing a refined owner in the return type: `bOwner`. As syntactic sugar, since the parametrization of the return type `A` of method `foo()` in class `C` is omitted, its owner is inferred based on the inheritance as `cOwner`. In contrast, the following definition is considered as invalid:

```
class D<dOwner> extends B<dOwner> {
  // ◁-contravariant method return type overriding
  A<?> foo(A<?> param) { ... } // error
}
```

There is no variance with respect to method parameter owners: the correspondence of owners of parameter types should be strict modulo class-specific substitution. If the overridden method's parameter type has the same class, but is *more defined than* ($\lhd$) the parameter type in the same method of the superclass, a necessary dynamic type

cast will be missing, which might lead to violation of the invariant, as in the following example:

```
class E<eOwner> extends B<dOwner> {
  // ◁-covariant parameter type overriding
  A<eOwner> foo(A<eOwner> param) { ... }
}
...
// p and q are arbitrary owners
A<p> e = new E<p>();
A<q> a1 = new A<q>();

// Type-correct, but missing
// a dynamic cast insertion to specify eOwner of a1
e.foo(a1);
```

In principle, we could allow one to use *less defined* owners in types of overridden methods (e.g., `?` instead of `eOwner`), as it would be consistent with the standard subtyping of functions, which assumes contravariance in parameter types. However, this would complicate the type checking procedure and conflict with the typing rules of Java 1.4, which does not allow contravariant method overriding, so we did not implement this option in the current prototype.

It is important to note that the variance issues described above are orthogonal to the variance of types with respect to nesting of *known* owners ($\prec$). Following the observation from the work of Clarke and Drossopoulou [33], we prohibit known owners to vary in corresponding types of subclasses, as this would make the type system unsound.

9.1.7 Current limitations

The current prototype implementation of gradual ownership types for Java 1.4 experiences a number of limitations with respect to language features. These limitations are the subject of the future work.

- Ownership parametrization of *arrays* is not supported. Intuitively, arrays as simple containers should be parametrized by two ownership parameters [170], in the spirit of the example with a single-linked list (Figure 7.1). However, doing so would require us to make changes in the representation of arrays on the level of a virtual machine, and we wanted to keep our translating compiler operating on the level of the source code.
- Current implementation does not allow one to define owner-polymorphic methods [37]. The system also does not make use of existential owners [26,

130]. However, the interplay between existential and gradual ownership types is beyond the scope of this work. The lack of owner-polymorphic methods and existential owners prevents from the smooth migration of the code implementing the logic of a *factory* pattern [90]. As a possible workaround, one can pass an additional immutable parameter to be used as a owner thanks to the dynamic aliasing. Of course, **`world`** is a valid owner in static methods.

9.2 Experience

> Key libraries for recurring data structures should be specifically studied in terms of their encapsulation properties and interconnection structure.
>
> TOBIAS WRIGSTAD and DAVE CLARKE [214]

How does one empirically validate a type system? A natural answer to this question is to use it for annotation and type-checking of a large existing codebase and measure how many components can be given types [202].

However, one more important aspect should be taken into account. Originally proposed gradual types by Siek and Taha [188] help to reduce the amount of casts in dynamically-typed languages. In contrast, gradual *ownership* types introduce *extra* dynamic checks, imposed by user-provided ownership annotations. As was shown at the end of Chapter 8, a fully-annotated program would not suffer from execution overhead. One may still wonder what the drawback in terms of performance is, when a program is annotated only partially, for instance, when only a minimal amount of annotations is provided in order to declare the ownership policy.

We provide quantitative answers to the questions about applicability of gradual ownership types and the performance overhead they impose by incrementally porting several classes from the Java Collection Framework (JCF) from Java SDK version 1.4.2 to use Gradual Ownership Types.

Most traditional collection classes that contain linked data structures implement internal logic to handle their entries in a way similar to the example in Figure 7.1. To design a case study for our experiment, we assumed that internal entries should be dominated by their outer collection instances, so they are not exposed to the external objects with the only exception of iterators [6, 23]. This makes collections with entries a good candidate for ownership types and the owners-as-dominators policy. Our intention was to ensure the OAD invariant holds for inner classes such as `Entry`

Class(es)	Files	LOC	Minimal annotations			Maximal annotations		
			Ann	Ratio	Ovhd	Ann	Ratio	Ovhd
`LinkedList`	6	2162	17	0.007	91%	34	0.015	0%
`TreeMap`	3	881	7	0.008	32%	28	0.03	15%
`LinkedList` `TreeMap`	8	3037	23	0.007	48%	61	0.02	8%

Figure 9.6: Results of annotating several classes from Java Collection Framework of Java SDK version 1.4.2 with gradual ownership types.

of collection classes such as `LinkedList` and `TreeMap`, without changing the existing code, but only by adding annotations.[2] The questions we were trying to answer are:

1. How many annotations (i.e., lines of code changed) are needed minimally to declare a desired policy with respect to the OAD invariant?
2. What is the execution overhead with added annotations in comparison with a program with no annotations at all?
3. How many annotations are needed for full static checking? and
4. Is it possible to fully migrate a set of classes (i.e., provide a static guarantee of invariant preservation) by adding annotations only?

Figure 9.6 provides a table with results of our experiments, reporting on the number of added annotations, affected lines of code and performance overhead, thereby answering the questions above.

The analysed code base consists of 46 source files, comprising about 8,200 lines of code, not including blank lines and comments. All these files were analysed by a compiler, however, only a few of them were actually changed by adding ownership annotations, as reported in the table with the results (the column "Files"). The experiments with strongly-encapsulated entries were made for two collection classes: `LinkedList` and `TreeMap`, separately and together. The first three columns provide a name of a collection class(es) instrumented with annotations, a number of affected files in a JCF (i.e., those requiring adding annotations), and the total number of lines

[2] A curious reader may wonder, why not provide ownership types to collection themselves, as, for example, in our running example with a single-linked list (Figure 7.1). We decided not to do so, as this migration would require recompiling the whole JCF library with new constructor signatures, which would make it unusable for any client code. In contrast, the changes required to encapsulate entries are rather modular.

of code in affected files. The next group of three columns presents the results for the *minimal* amount of annotations necessary to declare the encapsulation invariant. Column "Ann" provides the number of added annotations, Column "Ratio" represents a fraction Ann/LOC, and Column "Ovhd" reports on the *additional* execution overhead with respect to non-annotated code based on a series of extensive read/update benchmarks. The final group of three columns reports on the result of providing the *maximal* possible amount of annotations in order to get as close as possible to the static guarantee of the OAD invariant preservation.

Below, we discuss some important aspects of the migration, discovered while providing ownership annotations for collection classes `LinkedList` and `TreeMap`.

LinkedList

The minimal amount of annotations to ensure the OAD invariant for instances of the inner class `Entry` of `LinkedList` is 17, comprising 7 annotations to the `LinkedList` class itself and 10 in five other classes. Class `Iterator` was owner-parametrized to preserve the OAD invariant, as the inner class `ListItr` has access to entries of the list. The correctly annotated class `ListItr` is defined as follows; the iterator is owned by the instance of `LinkedList` (employing manifest ownership):

```
class ListItr implements ListIterator<LinkedList.this>
```

The ownership argument `LinkedList.this` stands for the reference to the outer instance of the inner class `ListItr`.

We implemented a series of simple benchmarks consisting mostly of multiple updates and iterations through a list. These reveal that the minimal annotations cause the average execution time per update to double (so the 91% overhead in Figure 9.6), mainly because of boundary checks involving use of reflection in order to examine the ownership structure of objects every time a list update was performed. However, the implementation of `LinkedList` in JCF allows *full* annotation. By adding 17 extra annotations in the `LinkedList` class (i.e., 34 annotations in total), one can reach zero execution overhead and full static preservation of the OAD invariant.

TreeMap

For the best result in terms of performance and the invariant preservation the `TreeMap` class requires 28 annotations, consisting of 26 annotations in the class itself and two extra annotations in the interfaces `Iterator` and `Map` respectively. Because of the static factory method `buildFromSorted`, which also operates on entries reconstructing a map, it is impossible to provide a fully static ownership

guarantee without modifying the original code. The possible solutions would be making the method non-static, or by providing an extra final method parameter as an alias for the potential owner, relying on the dynamic aliasing. Another alternative solution is to use owner-polymorphic methods [35], which are not supported in the current formalism. We tested the performance overhead by employing a set of stress benchmarks involving multiple updates and iterations. Because of incomplete annotation, in the presence of some non-avoidable casts for instances of entries created within the method `buildFromSorted`, the annotated `TreeMap` class still exhibits a small execution time overhead. This explains the 15% and 8% slowdowns reported in the last column of the table in Figure 9.6.

Detected possible object ownership leaks

Our compiler has helped to detect a place in the Java Collection Framework where a possible "leak" of the inner `Entry` classes with respect to the OAD invariant occurs. The class `ResourceBundleEnumeration` declares a package-protected field of type `Iterator`. Although this field is initialized with the iterator of some `Set` instance in the constructor, it can be reassigned elsewhere in client code, which may lead to an OAD invariant violation.

```
class ResourceBundleEnumeration implements Enumeration {

    Set set;
    Iterator iterator;
    Enumeration enumeration;

    ResourceBundleEnumeration(Set set, Enumeration enumeration) {
        this.set = set;
        this.iterator = set.iterator();
        this.enumeration = enumeration;
    }

    // other methods
}
```

In fact, this issue occurs because the relation between the objects referred by `set` and `iterator` is not captured in their type signatures. As a result, one can write the following code, which allows an arbitrary client of a `ResourceBundleEnumeration` instance to access the stored iterator, which clearly violates the OAD invariant:

```
Enumeration enum = ... ;
Set set = ... ;

ResourceBundleEnumeration rbe
    = new ResourceBundleEnumeration(set, enum);
Iterator wildIterator = rbe.iterator();
// use wildIterator
```

This problem could be remedied by making the field of type `Set` final and using it as a dynamic owner in the type of `iterator`. Our compiler would generate the code with necessary dynamic checks for updates of this field to enforce the invariant dynamically. However, to statically ensure OAD guarantee, significant refactoring would be required.

Concluding this chapter, we summarize that a relatively small number of necessary annotations and a reasonable performance overhead for non-fully annotated programs in our case study (Figure 9.6) make us believe that the current implementation of gradual ownership types can be applied to migrate large code bases to ownership types without significant changes in the code.

Chapter 10

Discussion and Related Work

In this chapter, we provide a discussion how to adapt the present approach for gradual ownership types to other ownership disciplines from the literature. We also provide a survey of related work, enumerating a series of techniques similar to the approach we have taken, namely, dynamic ownership and ownership inference.

10.1 Discussion

Several design choices were made in our approach to gradual ownership types. This section discusses other alternatives.

10.1.1 Alternative ownership disciplines

In our work we used the *owner-as-dominator* discipline as a base for applying the gradual technique. However, most of existing parametric ownership disciplines, such as *multiple ownership* [27], *ownership domains* [6], *external uniqueness* [35] or *owners-as-ombudsmen* [160], can be "gradualized" using a similar approach with *no changes* to the part related to type cast insertion. The difference between most of existing disciplines lies in the encapsulation invariant that is enforced and the relationships between owners and ownees that need to be maintained. In the present work, the encapsulation invariant is ensured using the boundary checks; other systems may require specific modifications of the definitions of the consistently nesting relation, the helper function **specified** and the runtime semantics of boundary checking according to the object encapsulation policy. For instance, in the case of ownership

domains [6] one can think of run-time checking whether an object belongs (perhaps, transitively) to a particular domain.

10.1.2 Required annotations and default conventions

The present approach required that ownership parameters be specified at all allocation sites. Hence object owners are all known at creation time. We decided on this requirement after analyzing existing approaches to ownership inference [92, 143]. All these approaches either require that some initial information on the ownership structure of objects is provided (i.e., the parametrization and the actual annotations) or they yield results that are hard to deal with (e.g., excessively large number of parameters per class or only trivial annotation schemes). The explanation of this phenomenon is simple: by specifying annotations, the programmer specifies her vision of the ownership structure of the object.

Two other possibilities were considered.

- The first approach was to annotate field and method types, thereby annotating the interface of the object. This approach unfortunately creates a significant overhead in the implementation, which would require run-time tracking of object aliasing: whenever an object owner becomes known, for example, by assignment into a field whose owners are specified, all other aliases to that object need to be checked for validity. Furthermore, the ownership of objects with the same owner as the assigned object also need to be updated—objects can have the same owner, even if this owner is not known; consider for example, the `Entry` objects in a linked list. The required run-time modifications are likely to introduce too much run-time overhead. This problem can be partially remedied by employing an advanced escape analysis [30], however, in the presence of multiple factory methods the impact of the analysis will be significantly reduced. Nevertheless, this approach is similar to what occurs in *dynamic ownership* [94].
- The second approach was to allow annotations to occur anywhere in the code. This approach is clearly best suited for programmers, but it clearly also suffers the same problems as annotating just the interface.

In principle, it is easy to implement a default convention following the design principle we have taken, according to which a objects instantiated at non-annotated allocation sites will be automatically owned by **`world`**. However, the current implementation does not feature this syntactic sugar.

10.1.3 Treatment of libraries

In order to ensure that library code correctly preserves the OAD invariant, our approach requires that certain classes be rewritten to enable client code to be sufficiently flexible. At the very least, a new parametrized definition of `Object` is required so that other classes can be parametrized. Our approach essentially assumes that any library code that needs to be owner-aware must be rewritten, but rewriting the library is a significant overhead, the kind which gradual typing aims to avoid.

Three alternative approaches to deal with libraries are possible.

- One is to ignore leaks of an object into ownership-unaware code, and assume a weaker ownership invariant that amounts to saying that an object is protected only within code compiled by our compiler, thinking of libraries as of "black holes". With this more pragmatic approach, library code can more gradually be converted to owner-aware code and trusted library code can "safely" be ignored. From the theoretical perspective this approach is promising, as one could elaborate a notion of *blame* [207] with respect to ownership types along with the corresponding principle *well-type code cannot be blamed for* OAD *violation*.
- A second alternative is to implement the byte-code instrumentation procedure that inserts the run-time checks to monitor field assignments in the code. In more detail, if an object is passed to a library, it should be passed in a way it will be never assigned to a field of an external (with respect to its owner) instance. Although this approach is feasible in practice, it is likely to introduce a big performance overhead, since any field update within libraries, possibly involving owned objects, should be checked in order not to violate the ownership invariant.
- The third approach is to perform a static analysis of library (byte)code along the lines of Ma and Foster's work [131] to infer possible ownership annotations for libraries. However, it might be hard to capture real ownership patterns by analyzing a standalone library with some assumptions about a context it might be used in, and the result of the analysis can be either too optimistic or too restrictive (see Section 10.2.4 for the survey of results in ownership inference).

10.1.4 Implementing boundary checks

In order to preserve the OAD invariant, boundary checks occur whenever an object is stored in a field of a type with an unknown primary owner of another object. An alternative look to a class field that does not specify its primary owner is that it

does not care what the owners are. One can thereby reformulate the OAD invariant (Definition 8.1.2) in a way that only owner-annotated fields participate in the definition. This would allow expensive boundary checks to be omitted, keeping only dynamic casts, at the expense of a weaker invariant. Such a system may be worth further investigation.

10.2 Related Work

Multiple approaches have been proposed in the last decade in order to bridge the gap beween the dynamically and statically typed programs. To the best of our knowledge, none were applied to ownership types. Below, we describe some of the "hybrid" approaches to type checking from the literature as well as provide a survey of relevant technique targeted to decrease verbosity of ownership types.

10.2.1 Gradual types and contracts

Our work is inspired by BabyJ language by Anderson and Drossopoulou [10], and is strongly based on the idea of gradual types by Siek and Taha [188, 189], which has been recently applied to Java-like generics [109] and modular typestate [210]. The notion of *blame control* is known in the context of gradual types to provide better debugging support [207]. Since dependent owners contain information about source code locations, the information from labels makes it easy to track back the flow dependencies and eliminate uncertainty by adding extra ownership annotations. This makes dependent owners similar to *blame labels*.

Like types by Wrigstad et al. [215] is a mechanism for the scripting language Thorn to provide static typing for dynamic values that exhibit a structure similar to the declared types. This structural treatment of like types allows a number of compiler optimization to be applied to the generated code. It is a matter of discussion whether like types can be easily extended to support enhanced object properties such as object ownership.

The idea of combining static and dynamic type checking is also close to the work of Flanagan on *hybrid types* [86]. Hybrid types may contain refinements in the form arbitrary predicates on underlying data. The type checker attempts to satisfy the predicates statically using a theorem prover. If the prover is not able to satisfy the predicate, the corresponding check is postponed until run-time.

Unlike the system we consider, hybrid types do *not* impose extra structural properties on the data they operate with. The run-time checker operates with type casts, treating them as logical implications, which makes it difficult to say if gradual ownership types can be formulated in terms of hybrid types without requiring necessary annotations

in order to specify the encapsulation boundaries. According to the classification proposed by Greenberg et al. [96], systems with hybrid and gradual types are related to the class *manifest* systems, i.e., those which contain additional constraints on data as part of types and enable a type checker to reason about them. Their counterparts are *latent* systems, in which contracts are purely dynamic checks [83]. Our framework is closer to the manifest systems because of type-level reasoning with owner parameters.

10.2.2 Dynamic ownership

Gordon and Noble, in their work on dynamic ownership, introduce ConstraintedJava, a scripting language that provides dynamic ownership checking [94]. The authors suggest a dynamic ownership structure consisting of an owner pointer in every object. Operations are provided to make use of and change these owner pointers. The semantics of the language relies on a message-passing protocol with a specific kind of monitoring, similar to our boundary checks. Messages are classified into several categories based on their relative positions of the message sender and receiver in the ownership tree. "Bad" messages are captured by run-time monitoring.

10.2.3 Existential types for ownership

Existential ownership types [130] offer variant subtyping of owners based on existential quantification [26]. This approach also allows *owner-polymorphic methods* to be elegantly implemented and it distinguishes objects with different and equal *unknown* owners. Existential quantification also helps to implement effective run-time downcasts in the presence of ownership types: a subtype's inferred owners are treated as existentially quantified [213]. The key difference between these approaches and ours is that existential ownership expresses *don't care* whereas gradual types express *don't know* concerning the unknown owners.

10.2.4 Ownership inference

The need for some sort of ownership inference as a way to decrease the annotation burden has been outlined in Clarke's PhD dissertation: *Ideally, we would like to specify only which objects are representation — that is have only the keyword* `rep` *— and let the compiler use static analysis to determine whether the representation really is protected* [37, page 188].

Ownership types systems generally require a significant amount of annotations to express the types, but this can be burdensome for the programmer. What makes

matters worse is that library code also needs to be annotated, in general, to work effectively with ownership types. Addressing this problem leads naturally to the question of ownership type inference. But matters are not so simple.

Unlike traditional type schemes à la Hindley-Milner, ownership annotations are mostly *design-driven*: it is mostly up to the programmer to decide whether some object should be owned by **`this`** or by **`world`**. Many ownership type systems admit a trivial collection of annotations, for example, by setting all objects to be owned by **`world`**. Consequently, even elaborate approaches to type qualifier inference [28, 97] are ineffective, as they infer any solution that satisfies the constraints, but cannot give a best, i.e., the most precise, solution.

In this section, we provide a brief survey of approaches for ownership inference, focusing mainly on those that infer type annotations, but also on those that infer properties of the ownership by examining the run-time structure of objects. Three approaches are considered: dynamic inference, static inference, and interactive inference.

Dynamic approaches

The pioneering work on the dynamic inference of ownership types is Wren's master's thesis [211]. The essence of his approach is to run programs with a profiler that keeps track of all heap "snapshots", collecting full information about the topology of the heap at any moment. All heap snapshots are then merged the resulting graph is analysed in order to infer dominance relations between objects. The work provides a graph-theoretical background for run-time inference, including a description of the most precise program heap topology with respect to the owners-as-dominators invariant. On the negative side, the ownership functions cannot be mapped directly to types. To remedy this, the author formulates the system of equations to assign annotations to particular object allocation sites. The proof of correctness of these equations as well as conditions for the uniqueness of the solution was left for future work.

Potanin, Noble and Biddle [169] employ a similar tool to take snapshots of the heaps of running programs. They applied their tool to a large corpus of Java programs and computed various metrics on the collected heaps related to notions of uniqueness, ownership and confinement, to determine how often such concepts appear in actual running programs. Their results indicate that such concepts are often used in practice. No indication was given on how to use the results to provide annotations to programs.

Mitchell provides a similar technique for dynamic ownership inference by summarizing memory footprints with help from the dominator relation [146]. In his approach, each dominator tree captures unique ownership. Trees are connected by specific edges

that capture *responsibility*, i.e., transfer of ownership. The profiling-based technique aggregate these structures, and use thresholds to identify important aggregates. The notion of ownership graph summarizes responsibility, and *backbone equivalence* aggregates patterns within trees, generating concise summaries of heap usage. The ultimate goal of this work is to understand where excessive memory usage occurs in large programs.

Dietl and Müller, in the first in the series of works towards a practical solution of the inference of Universe types [44], present results on runtime universe type inference [70]. As universe types require a comparatively lower annotation overhead, mapping of inference results to static annotations is easier than for the system Wren considered. The inference algorithm is, however, quite in the spirit of Wren's thesis: first a combined representation of the object store is built; then its dominator tree is constructed; finally, conflicts between the information obtained by analyzing the inferred dominator tree and the actual constraints of the type system are resolved by "flattening" the dominance trees and via a procedure the authors refer to as "harmonization". The resulting annotations deliver a correct typing of the program with respect to the target type system [44].

Static Approaches

One of the first attempts at providing ownership type inference was taken by Aldrich et al. [7]. In their system, the programmer needed only provide a small amount of annotations to indicate the intent that some parts of the program be encapsulated, and the rest of alias annotations were inferred. The approach was not entirely satisfactory, so subsequent work used more sophisticated analysis techniques, including variations of classic formulations of points-to or escape analysis, as well as incorporating elements of *may*- and *must*- abstractions [68, 112].

For instance, Moelius and Souter [147] employ a variation of an escape analysis [21] to infer ownership annotations. Their algorithm allows borrowed references to be returned from methods and assigned to object fields. No assumptions on ownership parameterization is made, and consequently the algorithm results in a large number of parameters.

For the same problem, Milanova and Liu [142] employ an Andersen-style points-to analysis [9] as part of static algorithm to infer ownership and universe annotations according to two different ownership protocols: owners-as-dominators and owners-as-modifiers. Both analyses are based on a context-insensitive points-to analysis, therefore they do not distinguish between different allocation and call sites. However, thanks to some Java-related heuristics, their technique handles some idiomatic cases, and good precision is thereby obtained. Their proof of correctness of the

Aspect	Gradual Ownership Types	Ownership Type Inference
Straightforward correspondence to the original type system	yes	no
Modular	yes	no
Effective debugging of type checking	yes	no
Well-typedness implies full static safety	no	yes
Minimal amount of annotations	required	optional
No runtime overhead	optional	yes

Figure 10.1: Comparison of Gradual Ownership Types and Ownership Type Inference by Huang and Milanova [104].

constructed dominance abstraction is present, although it does not rely on the abstract interpretation-like nature of the points-to analysis.

Later, Milanova and Vitek presented a static inference algorithm for ownership annotations for the owners-as-dominators invariant based on a static dominance inference algorithm [143]. The approach computes approximations of the object graphs using an enhanced *global* context-insensitive points-to analysis. The candidate ownership annotation are computed based on an approximated dominance tree, which is built based on a variation of *must-point-to* information. The approach does not provide any guarantee that the inferred annotations comply with the original ownership types system. In subsequent work employing the dominance inference algorithm, Huang and Milanova use the original type checker in order to verify the correctness of the inferred ownership annotations [104]. As our work is quite in the spirit of the results by Huang and Milanova, we provide a comparison of different aspects of Gradual Ownership Types and Ownership Type Inference in the table in Figure 10.1.

Inferring ownership as a property

In this section, we also mention several static analysis-based approaches that do not deliver type annotations directly as a result of inference, but extract encapsulation properties similar to those ensured by ownership type systems.

Geilmann and Poetzsch-Heffter [92] developed a modular abstract interpretation-based analysis to check simple (i.e., non-hierarchical) confinement properties in Java-like programs. This work employs a *box model* [168] instead of dominator trees. The approach is targeted to substitute modular type-checking by modular static analysis, requiring a significantly smaller amount of annotations: only class declarations and allocation sites need to be annotated. The analysis then takes the implementation of a class, considered as an encapsulated *box*, and executes it together with its *most-general client*. The most-general client is an abstraction of all possible clients which is used to create all possible traces through the box. If execution succeeds, the box never exposes any confined object, irrespective of the program that uses the box. The approach is based on formulating ownership as a *semantic* property of the program and the subsequent construction of the abstraction of the abstract semantics in the Cousot and Cousot's style [40, 41]. However, the complexity boundary of the derived analysis is unclear.

A general variation of a points-to analysis-based algorithm to infer ownership and uniqueness is presented by Ma and Foster [131]. The algorithm combines constraint-based intraprocedural and interprocedural analyses. The collected information about encapsulation properties is not however mapped to a type system. No correctness proof of the analysis is provided.

Interactive inference

Currently, *interactive approaches*, which require a small amount of interaction with the user when inferring ownership annotations, are one of the most promising directions in the solution of the ownership type inference problem.

Dietl et al. [69] presented a static analysis to infer Universe Types [44] according to the user-specified intentions, by solving a set of generated constraints. The first part of the technique is responsible for the generation of equations, based on the program semantics and the rules of the original type system. Constraints of the Universe Type system are encoded as a boolean satisfiability problem. The constraint-based analysis presented is close to traditional control-flow analyses via abstract interpretation. Once constraints are generated and solved, the second part of the approach comes to action in order to *tune* the result of the inference: users can indicate a preference for certain typings by adjusting the heuristics or by supplying partial annotations for the program. It has been empirically demonstrated that the NP-completeness of the constraint solving, reduced to the SAT problem, does not cause significant overhead on real-life problems as compared with other static approaches [104, 143, 147].

Two lines of research towards ownership type inference, via points-to analysis and via constraint solving, were unified in the work of Huang et al. [103]. The resulted framework implements checking and inference for two systems: Universe Types and

Ownership Types. As in prior work [69], the programmer can influence the inference by adding partial annotations to the program. In order to deliver the best solution of the inference, the authors formulate the *optimality* property with respect to introduced ordering on ownership annotations and user-provided input. The underlying analysis is implemented as a Kleene iteration of a monotonic transfer function, based on the program's small-step collecting semantics. The user-provided annotations are taken into account, whereas missing ones are initialized with the bottom element of the appropriate lattice.

Chapter 11

Conclusion and Future Work

11.1 Summary of Contributions

Introducing ownership types into real-life programs is a long-standing problem. The main causes are the verbosity of the formalism and its rigidity for some applications. In order to address these issues, in this work we applied the notion of gradual types to ownership type systems and the owners-as-dominators invariant for a Java-like language to seamlessly combine static and dynamic invariant checks. Our proposal distinguishes between the declarative part of the pluggable type system and helper annotations that can be omitted at the cost of dynamic checks. The developed framework has been formalized and proven to be correct. We implemented Gradual Ownership Types as an extension of an existing Java compiler and evaluated it on a well-studied codebase. With this work we also bring the notion of gradual types to the nominal type systems with additional structural properties, such as object ownership. The resulting approach enables the incremental migration from unannotated code to the code that uses ownership annotations and thereby preserves the invariant according to the programmer's intention. In addition, we believe that this work is a step towards the possible generalization of the idea of gradual types to pluggable type annotations in Java-like languages.

11.2 Future work

In the future work, we are going to investigate in detail type-theoretical aspects of gradual ownership types, focusing mainly on the role of dependent owners. We are

also going to explore the marriage of gradual types and static analyses for ownership inference and provide IDE support for the presented framework.

11.2.1 Gradual ownership types in higher-order languages

Although the ownership policy can be thought a specific kind of contract, which are widely used in some programming languages as a mechanism to ensure extra properties of data dynamically [83, 135], our calculus of gradual ownership types lacks the explicit notion of *blame labels* [5, 207]. However, the presence of dependent owners in our calculus partially remedies this lack, allowing the efficient tracking of origins of violation of invariants when a dynamic check fails. The usage of dependent owners for debugging ownership violations is a subject of a further investigation.

Also, in the future work we are going to extend the notion of gradual ownership types for higher-order languages in the spirit of Krishnaswami and Aldrich [119], unifying the notion of dependent owners and blame labels.

11.2.2 Gradual ownership types meet static analysis

It is tempting to use ownership annotations when performing static analysis of a program, for instance, for inferring may points-to invariants. Even a limited form of ownership annotations makes it possible to infer non-trivial properties of a global program heap topology using a simple instrumentation of a traditional store-based points-to analysis for object-oriented languages [185], making it possible to detect statically memory leaks and regions for safe garbage collection [29].

In their recent work on modular static analysis Tobin-Hochstadt and Van Horn propose the idea of using contracts as symbolic values to provide extra information for the static analysis about code whose implementation is missing (e.g., libraries) [203]. We believe, that using similar approach, even partial ownership annotations can be employed to leverage the points-to analysis via disjointness reasoning [33].

Moreover, by incorporating even a simple intraprocedural data-flow analysis into the framework of gradual ownership types, a fair amount of dynamic checks would be avoided. A variation of this optimization is discussed in Chapter 9 (Section 9.1.3). We leave a detailed investigation of the interplay between gradual ownership types and static analyses for future work.

11.2.3 IDE support

The current Gradual Ownership Framework for Java 1.4 has been implemented using the JastAddJ framework [74], which makes us believe that the adoption of our extended type system and a translating compiler is feasible with no big effort. In future work, we plan to provide JastAdd-based IDE support to help the migration from ownership-unaware code to fully owner-annotated code. Our belief is supported by recent results employing Reference Attribute Grammars [73] as formal specifications for generating basic functionality of a language-specific integrated development environment [193], namely, static error highlighting and reference resolution.

In the case of our framework, it should be possible to reuse the logic of the translating compiler for emitting error messages every time a type checking error has occurred and warning messages when the equality of owners cannot be proved statically, providing a programmer with good level of assistance when adding missing ownership annotations to the code.

Concluding Words

In this dissertation we focused on two problems concerning type systems in programming languages.

In the first research track, we presented a motivation to employ operational formalisms to define algorithms for type checking, arguing that such an operational view on type checking procedures gives a solid basis for better understanding and debugging of type systems in practice. As two main examples, we considered a reduction semantics for type checking and type checking in the form of an abstract machine. The problem we addressed in this part of the work is a proof of correspondence between different type-checking semantics. Thinking of the semantics in terms of the corresponding *interpreters*, we have applied a program transformation tool-chain, known as *functional correspondence*, in order to inter-derive formalisms, rather than construct them from scratch and then prove the corresponding equivalence theorems. As a main result of this work, we have provided a methodology to build families of type checkers equivalent *by construction*.

In the second part of the work, we focused on the problem of verbosity of annotations and the rigidity of static restrictions in advanced type systems for object-oriented languages. We have taken a type system ensuring a strong encapsulation property in a Java-like language and described a way in which it can be made *gradual*, i.e., incorporating both static and dynamic checks and giving a freedom to the programmer to choose between them. Our approach provides a tradeoff between the annotation burden and the performance drawback dynamic invariant checking.

We believe that together the two parts of our work make steps towards developing approaches to ensure desired properties of programs, abstracting over property-checking algorithms and type annotation policies.

Appendix A

PLT Redex Implementation of Type Checking Semantics

In this appendix, we provide the full implementation of both operational semantics for type checking: the reduction one and the one in the form of SEC machine (Section 1.1). Both semantics are implemented in the PLT Redex framework [79]. The implementations are directly executable and were used to generate the figures in Chapter 1, Section 1.1. The code is also available from GitHub:

`http://github.com/ilyasergey/typechecker-transformations`

A.1 An Implementation of Type Checking with Reductions

plt-redex/type-reduction.rkt

```
#lang racket

(require redex)

;; Syntax
(define-language λrt
  ; Expressions
  [e n x (λ (x τ) e) (e e) (-> τ e) num]

  ; Numbers
  [n number]
```

```
  ; Contexts
  [T (T e) (τ T) (-> τ T) hole]
  ; Types
  [τ num (-> τ τ)]
  ; Variables
  [x variable-not-otherwise-mentioned])

;; Contractions
(define t-red
  (reduction-relation
   λrt
   (--> (in-hole T n)
        (in-hole T num)
        "[tc-const]")
   (--> (in-hole T (λ (x τ) e))
        (in-hole T (-> τ (subst (x τ) e)))
        "[tc-lam]")
   (--> (in-hole T ((-> τ_1 τ_2) τ_1))
        (in-hole T τ_2)
        "[tc-τβ]")))

;; Substitution does not need to be capture-avoiding
;; in the case of λrt since variable and type names are
;; in different semantic spaces
(define-metafunction λrt
  subst : (x any) any -> any
  ;; 1. x_1 bound, so don't continue in lambda body
  [(subst (x_1 any_1) (λ (x_1 τ_1) any_2))
   (λ (x_1 τ_1) any_2)]
  ;; 1. replace x_1 with e_1
  [(subst( x_1 any_1) x_1) any_1]
  ;; 2. x_1 and x_2 are different, so don't replace
  [(subst (x_1 any_1) x_2) x_2]
  ;; the last cases cover all other expressions
  [(subst (x_1 any_1) (any_2 ...))
   ((subst (x_1 any_1) any_2) ...)]
  [(subst (x_1 any_1) any_2) any_2])

;; type? : hybrid-expression -> boolean
;; A predicate to check if is a type
(define type? (redex-match λrt τ))

;; single-step? : expression -> boolean
(define (single-step? e)
  (= (length (apply-reduction-relation t-red e)) 1))

;; General well-formedness predicate
(define (is-ok? e)
  (or (type? e) (single-step? e)))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Examples
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define example-wt
  (term (λ (y (-> num (-> num num)))
           (λ (x num) (y x)))))

(define example-ill
  (term (λ (x (-> num num))
           (λ (y num) ((x y)
                       (λ (z num) (x z)))))))
```

Figure 1.2 from Section 1.1 was obtained by executing the following command after the file `type-reduction.rkt` was loaded:

```
;; Generate Figure 1.2
(traces t-red example-ill #:pred is-ok?)
```

A.2 Implementation of Type-Checking via the SEC Machine

plt-redex/type-sec-machine.rkt

```
#lang racket

(require redex)

;; Syntax
(define-language λsec
  ; Expressions
  (e n x (λ (x τ) e) (e e))
  ; Numbers
  (n number)
  ; Types
  (τ num (-> τ τ))
  ; Environments
  (E ((x τ) ...))
  ; Variables
  (x variable-not-otherwise-mentioned)
  ; Result stacks
  (S (τ S) nil)
  ; Control elements
  (c e (Lam τ S) (Fun e) (Arg τ_1 τ_2))
  ; Control stacks
  (C (c C) nil)
  ; States
  (ξ (S E C)))
```

```
;; Small-step abstract machine
(define t-sec
  (reduction-relation
   λsec #:domain ξ
   (--> (S E (n C))
        ((num S) E C)
        "[num]")
   (--> (S E (x C))
        (((env-lookup E x) S) E C)
        "[var]")
   (--> (S E ((λ (x τ) e) C))
        (nil (env-extend E (x τ)) (e ((Lam τ S) C)))
        "[lam]")
   (--> (S E ((e_1 e_2) C))
        (S E (e_1 ((Fun e_2) C)))
        "[app]")
   (--> ((τ_2 S) E ((Lam τ_1 S_1) C))
        (((-> τ_1 τ_2) S_1) E C)
        "[t-lam]")
   (--> (((-> τ_1 τ_2) S) E ((Fun e_2) C))
        (((-> τ_1 τ_2) S) E (e_2 ((Arg τ_1 τ_2) C)))
        "[t-fun]")
   (--> ((τ_1 (any S)) E ((Arg τ_1 τ_2) C))
        ((τ_2 S) E C)
        "[t-arg]")))

;; Environment lookup
(define-metafunction λsec
  env-lookup : E x -> τ
  [(env-lookup ((x τ) (x_1 τ_1) ...) x) τ]
  [(env-lookup ((x_1 τ_1) (x_2 τ_2) ...) x)
   (env-lookup ((x_2 τ_2) ...) x)])

;; Environment extension
(define-metafunction λsec
  env-extend : E (x τ) -> E
  [(env-extend ((x_1 τ_1) ...) (x τ))
   ((x τ) (x_1 τ_1) ...)])

;; Inject expression into a machine state
(define-metafunction λsec
  inject : e -> ξ
  [(inject e) (nil () (e nil))])

;; single-step? : expression -> boolean
(define (single-step? s)
  (= (length (apply-reduction-relation t-sec s)) 1))

;; final-state? : state -> boolean
(define (final-state? ξ)
  (eq? 'nil (caddr ξ)))
```

```
;; General well-formedness predicate
(define (is-ok? ξ)
  (or (final-state? ξ) (single-step? ξ)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Examples
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define example-wt
  (term (inject (λ (y (-> num (-> num num)))
                  (λ (x num) (y x))))))

(define example-ill
  (term (λ (x (-> num num))
          (λ (y num) ((x y)
                      (λ (z num) (x z)))))))
```

Figure 1.3 from Section 1.1 was obtained by executing the following command after the file `type-sec-machine.rkt` was loaded:

```
;; Generate Figure 1.3
(traces t-sec example-ill #:pred is-ok?)
```

Appendix B

Initial Function Definitions of Chapter 3

This appendix provides a detailed overview of some missing functions from the implementation of the original reduction semantics for type checking. The code in this appendix is a starting point for transformations described in Chapter 3 The full code of derivations is available at GitHub:

`http://github.com/ilyasergey/typechecker-transformations`

B.1 Plain Syntax Implementation

This section describes elements of the syntax of the traditional λ-calculus. The abstract syntax for $\lambda_{\mathcal{H}}$ includes integer literals, identifiers, lambda-abstractions, applications as well as "hybrid" elements such as numeric types and arrows $\tau \to e$. Types are either numeric types or arrow types. The special value `T_ERROR s` is used for typing errors; it cannot be a constituent of any other type. Typing environments `TEnv` represent bindings of identifiers to types, which are values in the hybrid language.

syntax.sml

```
structure Syn =
struct
datatype typ = T_NUM
             | T_ARR of typ * typ
             | T_ERROR of string

datatype term = LIT of int
              | IDE of string
              | LAM of string * typ * term
              | APP of term * term

end

structure TEnv =
struct
type 'a gamma = (string * 'a) list

val empty = []

fun extend (x, t, gamma) = (x, t) :: gamma

fun lookup (x, gamma)
    = let fun search []
              = NONE
            | search ((x', t) :: gamma)
              = if x = x' then SOME t else search gamma
      in search gamma
      end
end
```

B.2 Hybrid Syntax Implementation

We introduce closures into the hybrid language in order to represent the environment-based reduction system. A closure can either be a number, a ground closure pairing a term and an environment, a combination of closures, a closure for a hybrid arrow expression, or a closure for a value arrow element, namely an arrow type. A value in the hybrid language is either an integer or a function type. Environments bind identifiers to values.

hsyntax.sml

```
(* Hybrid syntax *)
structure HSyn =
struct
open Syn TEnv

datatype hterm = H_LIT of int
               | H_IDE of string
               | H_LAM of string * typ * hterm
               | H_APP of hterm * hterm
               | H_TARR of typ * hterm
               | H_TNUM

datatype closure = CLO_NUM
                 | CLO_GND of hterm * bindings
                 | CLO_APP of closure * closure
                 | CLO_ARR of typ * closure
                 | CLO_ARR_TYPE of typ
withtype bindings = typ TEnv.gamma

datatype hctx = CTX_MT
              | CTX_FUN of hctx * closure
              | CTX_ARG of typ * hctx
              | CTX_ARR of typ * hctx
end
```

B.3 Implementation of the Reduction Semantics for Type Checking

This section provides full implementation of the original reduction semantics for type checking operating with elements of both plain and hybrid syntax.

reductions.sml

```
use "syntax.sml";
use "hsyntax.sml";

structure TypeCheck_Reduct =
struct

open HSyn TEnv

fun type_to_closure T_NUM
    = CLO_NUM
  | type_to_closure (v as T_ARR (t1, t2))
    = CLO_ARR_TYPE v

fun term_to_hterm (IDE s)
```

```
    = H_IDE s
  | term_to_hterm (LAM (x, t, e))
    = H_LAM (x, t, term_to_hterm(e))
  | term_to_hterm (LIT i)
    = H_LIT i
  | term_to_hterm (APP (e1, e2))
    = H_APP (term_to_hterm e1, term_to_hterm e2)


datatype potential_redex = PR_NUM
                         | PR_LAM of string * typ * hterm * bindings
                         | PR_APP of typ * typ
                         | PR_ARR of typ * typ
                         | PR_IDE of string * bindings
                         | PR_PROP of hterm * hterm * bindings

datatype contractum_or_error = CONTRACTUM of closure
                             | ERROR of string
(*  contract : potential_redex -> contractum_or_error  *)
fun contract PR_NUM
    = CONTRACTUM CLO_NUM
  | contract (PR_ARR (t1, t2))
    = CONTRACTUM (type_to_closure (T_ARR (t1, t2)))
  | contract (PR_IDE (x, bs))
    = (case TEnv.lookup (x, bs)
        of NONE => ERROR "undeclared identifier"
         | (SOME v) => CONTRACTUM (type_to_closure v))
  | contract (PR_LAM (x, t, e, bs))
    = CONTRACTUM (CLO_GND (H_TARR (t, e), TEnv.extend (x, t, bs)))
  | contract (PR_APP (T_ARR (t1, t2), v))
    = if t1 = v
      then CONTRACTUM (type_to_closure t2)
      else ERROR "parameter type mismatch"
  | contract (PR_PROP (t0, t1, bs))
    = CONTRACTUM (CLO_APP (CLO_GND (t0, bs), CLO_GND (t1, bs)))
  | contract (PR_APP (t1, t2))
    = ERROR "non-function application"

datatype type_or_decomposition = VAL of typ
                               | DEC of potential_redex * hctx

(*  decompose_closure : closure * hctx -> type_or_decomposition *)
fun decompose_closure (CLO_NUM, C)
    = decompose_context (C, T_NUM)
  | decompose_closure (CLO_ARR_TYPE v, C)
    = decompose_context (C, v)
  | decompose_closure (CLO_GND (H_LIT n, bs), C)
    = decompose_context (C, T_NUM)
  | decompose_closure (CLO_GND (H_IDE x, bs), C)
    = DEC (PR_IDE (x, bs), C)
  | decompose_closure (CLO_GND (H_LAM (x, t, e), bs), C)
    = DEC (PR_LAM (x, t, e, bs), C)
  | decompose_closure (CLO_GND (H_APP (t0, t1), bs), C)
```

```
    = DEC (PR_PROP (t0, t1, bs), C)
  | decompose_closure (CLO_GND (H_TNUM, bs), C)
    = decompose_context (C, T_NUM)
  | decompose_closure (CLO_GND (H_TARR (t, e), bs), C)
    = decompose_closure (CLO_GND (e, bs),
                         CTX_ARR (t, C))
  | decompose_closure (CLO_APP (c0, c1), C)
    = decompose_closure (c0, CTX_FUN (C, c1))
  | decompose_closure (CLO_ARR (v, c), C)
    = decompose_closure (c, CTX_ARR (v, C))

(*  decompose_context : hctx * typ -> type_or_decomposition *)
and decompose_context (CTX_MT, v)
    = VAL v
  | decompose_context (CTX_FUN (C, c1), v0)
    = decompose_closure (c1, CTX_ARG (v0, C))
  | decompose_context (CTX_ARG (v0, C), v1)
    = DEC (PR_APP (v0, v1), C)
  | decompose_context (CTX_ARR (v0, C), v1)
    = DEC (PR_ARR (v0, v1), C)

(*  decompose : closure -> type_or_decomposition *)
fun decompose c
  = decompose_closure (c, CTX_MT)

(*  recompose : hctx * closure -> closure *)
fun recompose (CTX_MT, c)
    = c
  | recompose (CTX_FUN (C, c1), c0)
    = recompose (C, CLO_APP (c0, c1))
  | recompose (CTX_ARG (v0, C), c1)
    = recompose (C, CLO_APP (type_to_closure v0, c1))
  | recompose (CTX_ARR (v0, C), c1)
    = recompose (C, CLO_ARR (v0, c1))

datatype result = RESULT of typ
                | WRONG of string

(*  iterate : type_or_decomposition -> result *)
fun iterate (VAL v)
    = RESULT v
  | iterate (DEC (pr, C))
    = (case contract pr
        of (CONTRACTUM c')
           => iterate (decompose (recompose (C, c')))
         | (ERROR s)
           => WRONG s)

(*  normalize : term -> result *)
fun normalize t
    = iterate (decompose (CLO_GND (term_to_hterm t, TEnv.empty)))

(*  type_check : term -> typ *)
```

```
fun type_check t
    = case normalize t
       of (RESULT v)
          => v
        | WRONG s
          => T_ERROR s
end
```

Appendix C

Proofs from Chapter 8

> Unlike proofs in pure mathematics, proofs in computer science are usually tedious, inelegant and are worth something only because of their existence.
>
> GREGORY MORRISETT

In this appendix, we provide detailed proofs of main statements about type-based compilation from the language $\mathsf{JO}_?$ to the language $\mathsf{JO}_?^+$ (Section C.1) and type safety properties of the languages $\mathsf{JO}_?/\mathsf{JO}_?^+$ (Section C.2).

C.1 Proofs about compilation (Section 8.2)

In this section, we provide proofs of main lemmas from Section 8.2, relating well-typed programs in the calculus of gradual ownership types $\mathsf{JO}_?$ and appropriate translations: type cast and boundary check insertions. Theorem 8.2.12, a corollary of these lemmas, establishes one of the main result of this work: *well-typed* programs with gradual types are always translated to *well-typed* programs with inserted checks.

Lemma C.1.1 ($\overset{c}{\rightsquigarrow}$ is $\vdash^c$-sound **(Lemma 8.2.6)**). *If* $E \vdash e \overset{c}{\rightsquigarrow} e' : s$ *then* $E \vdash^c e' : s$.

Proof. The proof is by induction on the cast insertion derivation, i.e., the "depth" of the processed expression. The base of induction is trivial, since the expressions consisting from only one variable are not affected by the translation, and their typing rules in $\vdash^c$ are similar to the rules of $\vdash$. The same reasoning is applied to the

expressions of the kind $\texttt{let}\ x = b\ \texttt{in}\ e$, where either $b = \texttt{new}\ \ c\langle r_{i\in 1..n}\rangle$ for some c, or $b = \texttt{null}$, or $b = y.f$ for some y, f, because the translation $\overset{\mathcal{C}}{\rightsquigarrow}$ does not change these expressions and type rules in $\vdash$ and $\vdash^{\mathcal{C}}$ are similar. Only two kinds of expressions we need to consider are *field updates* and *method calls*.

Case (C-UPD)**.** Assume

$$E \vdash \texttt{let}\ x = (z.f = y)\ \texttt{in}\ e_1 \overset{\mathcal{C}}{\rightsquigarrow} \mathcal{C}_E\langle s, \sigma_z(t)\rangle(\texttt{let}\ x = (z.f = y)\ \texttt{in}\ e_2) : s'$$

1. By assumption, obtain
   (a) $E \vdash z : c\langle\sigma\rangle$
   (b) $\mathcal{F}_c(f) = t$
   (c) $E \vdash y : s$
   (d) $E \vdash s \lesssim \sigma_z(t)$
   (e) $E, x : \mathsf{fill}(x, \sigma_z(t)) \vdash e_1 \overset{\mathcal{C}}{\rightsquigarrow} e_2 : s'$
2. By the definition of $\mathcal{C}$ (Figure 8.9), we consider two cases
   (a) $E \vdash s \lhd \sigma_z(t)$, then $\mathcal{C}$ is an identity function and the typing rule (T-UPD') is directly applicable, so the further proof is trivial.
   (b) $E \vdash s \not\lhd\ \sigma_z(t)$, then
   $$\begin{array}{rl} & \mathcal{C}_E\langle s, \sigma_z(t)\rangle(\texttt{let}\ y' = (z.f = y)\ \texttt{in}\ e_2) \\ \equiv & \texttt{let}\ y' = \langle\sigma_z(t)\rangle y\ \texttt{in}\ (\texttt{let}\ x = (z.f = y')\ \texttt{in}\ e_2). \end{array}$$
3. By induction hypothesis and (1:e), obtain $E, x : \mathsf{fill}(x, \sigma_z(t)) \vdash^{\mathcal{C}} e_2 : s'$.
4. By the rule (T-CAST), (1:c), (1:d), obtain $E \vdash^{\mathcal{C}} \langle\sigma_z(t)\rangle y : \sigma_z(t)$.
5. Assuming $E' = E, y' : \mathsf{fill}(y', \sigma_z(t))$, by (T-UPD'), obtain $E \vdash^{\mathcal{C}} (z.f = y') : \sigma_z(t)$.
6. By (T-LET), (3) and (5), obtain $E' \vdash^{\mathcal{C}} \texttt{let}\ x = (z.f = y')\ \texttt{in}\ e_2 : s'$.
7. By (T-LET), (4) and (6), obtain $E \vdash^{\mathcal{C}} \texttt{let}\ y' = \langle\sigma_z(t)\rangle y\ \texttt{in}\ \texttt{let}\ x = (z.f = y')\ \texttt{in}\ e_2 : s'$, which concludes the proof for this case. Note that the environment E' has turned to E thank to the explicit introduction of the variable y' in the resulting `let`-expression.

Case (C-CALL)**.** Assume

$$E \vdash \texttt{let}\ x = z.m(y)\ \texttt{in}\ e_1 \overset{\mathcal{C}}{\rightsquigarrow} \mathcal{C}_E\langle s, \sigma_z(t)\rangle(\texttt{let}\ x = z.m(y)\ \texttt{in}\ e_2) : s'$$

1. By assumption, obtain

(a) $E \vdash z : c\langle\sigma\rangle$

(b) $\mathcal{M}\,\mathcal{T}_c(m) = (y'', t \to t')$

(c) $E \vdash y : s$

(d) $E \vdash s \lesssim \sigma_z(t)$

(e) $\sigma' \equiv \sigma \uplus \{y'' \mapsto y\}$

(f) $E, x : \mathsf{fill}(x, \sigma'_z(t')) \vdash e_1 \overset{\mathcal{C}}{\rightsquigarrow} e_2 : s'$

2. By the definition of $\mathcal{C}$ (Figure 8.9), we consider two cases

   (a) $E \vdash s \lhd \sigma_z(t)$, then $\mathcal{C}$ is an identity function and the typing rule (T-CALL') is directly applicable, so the further proof is trivial.

   (b) $E \vdash s \not\lhd \sigma_z(t)$, then

   $$\mathcal{C}_E\langle s, \sigma_z(t)\rangle(\texttt{let}\ y' = z.m(y)\ \texttt{in}\ e_2) \equiv \texttt{let}\ y' = \langle\sigma_z(t)\rangle y\ \texttt{in let}\ x = z.m(y')\ \texttt{in}\ e_2$$

3. By induction hypothesis and (1:e), obtain $E, x : \mathsf{fill}(x, \sigma'_z(t')) \vdash^c e_2 : s'$.

4. By the rule (T-CAST), (1:c), (1:d), obtain $E \vdash^c \langle\sigma_z(t)\rangle y : \sigma_z(t)$.

5. Assuming $E' = E, y' : \mathsf{fill}(y', \sigma_z(t))$, by (T-CALL'), obtain $E' \vdash^c z.m(y') : \sigma'_z(t')$.

6. By (T-LET), (3) and (5), obtain $E' \vdash^c \texttt{let}\ x = z.m(y')\ \texttt{in}\ e_2 : s'$.

7. By (T-LET), (4) and (6), obtain $E \vdash^c \texttt{let}\ y' = \langle\sigma_z(t)\rangle y\ \texttt{in let}\ x = z.m(y')\ \texttt{in}\ e_2 : s'$, which concludes the proof for this case.

□

Corollary C.1.2 ($\overset{\mathcal{C}}{\rightsquigarrow}$ is $\vdash^c$-sound for methods **(Corollary 8.2.7)**). *If* $E \vdash t'\ m(t\ y)\ \{e\} \overset{\mathcal{C}}{\rightsquigarrow} t'\ m(t\ y)\ \{e'\}$ *then* $E \vdash^c t'\ m(t\ y)\ \{e'\}$

Proof. Assume

$$E \vdash t'\ m(t\ y)\ \{e\} \overset{\mathcal{C}}{\rightsquigarrow} t'\ m(t\ y)\ \{F[\mathcal{C}_E\langle s, t'\rangle(z)]\},\ \text{where}$$

(i) $E \vdash e : s$

(ii) $E \vdash s \lesssim t'$

(iii) $E, y : \mathsf{fill}(y, t) \vdash e \overset{\mathcal{C}}{\rightsquigarrow} e' : s$

(iv) $e' = F[z]$

We will prove an auxiliary statement:

Proposition C.1.3. *If $E \vdash s \lesssim t'$, $e' = F[z]$ and $E \vdash^c e' : s$ then $E \vdash^c F[\mathcal{C}_E\langle s,t'\rangle(z)] : s'$ such that either $E \vdash s' = t'$ or $E \vdash s' = s$ and $E \vdash s \lhd t'$.*

Proof. The proof is by induction on the depth of the reduction context F.

Case ($F \equiv [\,]$ (Induction base))**.** By assumption, $e' = z$

1. By the definition of $\mathcal{C}$ (Figure 8.9), we consider two cases
   (a) $E \vdash s \lhd t'$, then $\mathcal{C}$ is an identity function and the typing rule (T-CALL') is directly applicable, so the further proof is trivial, as we have $E \vdash s' = s$.
   (b) $E \vdash s \not\lhd t'$, then $\mathcal{C}_E\langle s,t'\rangle(z) \equiv \texttt{let}\ z' = \langle t'\rangle z\ \texttt{in}\ z'$
2. By (T-CAST), (T-LET) and (1:b), obtain $E \vdash^c \texttt{let}\ z' = \langle t'\rangle z\ \texttt{in}\ z' : t'$. Equivalently, $E \vdash^c F[\mathcal{C}_E\langle s,t'\rangle(z)] : t'$ and $E \vdash s' = t'$.

Case ($F \equiv \texttt{let}\ x = b\ \texttt{in}\ F'$)**.** The reasoning is by induction hypothesis.

1. By assumption $E \vdash^c F[z] : s$, where $F[z] = e'$.
2. By (T-LET), obtain
   (a) $E \vdash^c b : t$ for some t
   (b) $E' \vdash^c F'[z] : s$, where $E' = E, x : \mathsf{fill}(x,t)$
3. By induction hypothesis and (2:b), obtain
   (a) $E' \vdash^c F'[\mathcal{C}_{E'}\langle s,t'\rangle(z)] : s'$ for some s'
   (b) $E' \vdash s' = t'$ or $E' \vdash s' = s \ \wedge\ E' \vdash s \lhd t'$
4. Since $E \vdash^c e' : s$ implies $E \vdash s$, we may conclude that there is no occurrences of the variable x as an owner parameter in the type s (but there might be some $x^{c.i}$, which is handled by the rule (OWN-DEPENDENT)). There are also no occurrences of x in t, so we can reduce E' to E and both types will remain *well-formed* in E and so will s'.
5. By (T-LET) and (3:a), obtain $E \vdash^c \texttt{let}\ x = b\ \texttt{in}\ F'[\mathcal{C}_E\langle s,t'\rangle(z)] : s'$
6. By (3:a), (3:b) and (4), obtain
   (a) $E \vdash^c F[\mathcal{C}_E\langle s,t'\rangle(z)] : s'$
   (b) $E \vdash s' = t'$ or $E' \vdash s' = s \ \wedge\ E \vdash s \lhd t'$

□

The proof of the enclosing corollary follows straightforwardly from the proved proposition via initial assumptions of (()c-method) and the rule for well-formed methods (METHOD'). □

Lemma C.1.4 ($\overset{\mathcal{B}}{\rightsquigarrow}$ is $\vdash^{\mathcal{C}}_{\mathcal{B}}$-sound **(Lemma 8.2.9)**). *If* $E \vdash e' \overset{\mathcal{B}}{\rightsquigarrow} e'' : s$ *then* $E \vdash^{\mathcal{C}}_{\mathcal{B}} e'' : s$.

Proof. The proof is by induction on the boundary check insertion derivation. The only one kind of expressions that should be considered is those that contain field assignment as an underlying statement, i.e. $\texttt{let}\ x = (z.f = y)\ \texttt{in}\ e$. By the definition of $\mathcal{B}$ (Figure 8.9), there are two possible cases: when the owner of a type t is specified or not. In the first case, $\mathcal{B}$ is just an identity translation, and the transition from (T-UPD') to (T-UPD") is straightforward. Otherwise, the statement $z.f = y$ is replaced by $z.f \leftarrow y$, and the rule (T-CHECK) is applied for typing. □

Corollary C.1.5 ($\overset{\mathcal{B}}{\rightsquigarrow}$ is $\vdash^{\mathcal{C}}_{\mathcal{B}}$-sound for methods **(Corollary 8.2.10)**). *If* $E \vdash t'\ m(t\ y)\ \{e\} \overset{\mathcal{C}}{\rightsquigarrow} t'\ m(t\ y)\ \{e'\}$ *then* $E \vdash^{\mathcal{C}} t'\ m(t\ y)\ \{e'\}$

Proof. The proof is the similar to the one of Corollary C.1.2. □

C.2 Proofs about Type Safety and Invariant Preservation (Section 8.4)

This section provides proofs for the main results of the Chapter 8: subject reduction and OAD preservation theorems from Section 8.4 (Theorems 8.4.6 and 8.4.7 respectively). The employed techniques are standard. We also provide an amount of binding remarks to indicate the general flow of results and reasonings.

Remark C.2.1. In the statement of Theorem 8.4.6 we assume all ownership arguments of involved types t, s etc. to be actual (see Figure 8.2), i.e., some *heap locations* or world. When reasoning about typings in the presence of local typing binding environments we exploit the equalities, provided by the rules (IN-BIND1), (IN-BIND2) and (IN-BIND3). Proposition C.2.2 and Corollary C.2.3 formalize this observation. The statement of Lemma C.2.7 brings the same equivalence to the *type instantiation* relation ($\ltimes$).

Proposition C.2.2. *If* $\mathcal{E}, E; B \vdash t$ *for some* $t = c\langle p_{i\in 1..n}\rangle$ *and* $\mathrm{dom}(E) = \mathrm{dom}(B)$ *then there exists* $t' = c\langle q_{i\in 1..n}\rangle$, *such that* $\mathcal{E}, E; B \vdash t = t'$ *and for all i,* $q_i =?$ *or* $\mathsf{actual}(q_i)$.

Proof. The proof is by the fact that $\mathcal{E}, E; B \vdash \diamond$. The type t' is constructed via the rules (IN-BIND*). □

Corollary C.2.3. *If* $\mathcal{E}, E; B \vdash t$ *for some* t *and* $\mathrm{dom}(E) = \mathrm{dom}(B)$ *then there exists* t'*, such that* $\mathcal{E}, E; B \vdash t = t'$ *and* $\mathcal{E} \vdash t'$.

Proof. The type t' is the one built in Proposition C.2.2. Since the structure of the type t' does not contain any components but actual owners or ?, the environments E and B do not contribute to its structural properties, so can be excluded from the ultimate typing judgement. □

In the course of the following proofs it is important to distinguish with *syntactic* types, possibly having unknown owners as their constituents, and "unequivocal" types, where unknown owners are eliminated by replacing them either with known, formal or dependent owners. We formalize this distinction by the following definition:

Definition C.2.4. The type $t = c\langle p_{i \in 1..n} \rangle$ is *unequivocal* iff $\forall i \in 1..n,\ p_i \neq ?$.

We will also refer to non-unknown owners as to *unequivocal* ones.

Remark C.2.5. The useful property of the typing relations $\vdash$ and, consequently, $\vdash^{c}_{\mathcal{B}}$ and $\Vdash$ is that the resulting type, assigned to an *expression* (but not a *statement*!) is a *unequivocal* type.

We formalize this observation as a following lemma:

Lemma C.2.6. *If* $E; B \mathrel{\mathcal{R}} e : t$ *for some* e *and* $\mathcal{R} \in \{\vdash, \vdash^{c}, \vdash^{c}_{\mathcal{B}}\}$ *then* t *is unequivocal.*

Proof. On the structure of typing rules and the definition of the helper function fill. □

The following lemma states an important fact about the type instantiation as a bounding chain between syntactic and run-time types.

Lemma C.2.7 (Type instantiation)**.**

$$\left.\begin{array}{l} \mathcal{E}, E; B \vdash t \\ \mathcal{E} \vdash H \\ H; B \vdash t \ltimes t' \end{array}\right\} \quad \Rightarrow \quad \mathcal{E}, E; B \vdash t = t'$$

Proof. The proof is by Definition 8.3.1. Two types t and t' are equal, if under provided assumptions on variables, types and owners $\mathcal{E}, E; B$, one can prove the equality of owner constituents of t and t' respectively. Assume $t = c\langle p_{i \in 1..n} \rangle$ and $t' = c\langle q_{i \in 1..n} \rangle$. Let us consider all possible kinds of p_i in t.

Case $(p_i = x^{c.j})$**.** $H(B(x)) = \langle t, \ldots \rangle$ and $k_j = \mathsf{owner}_j(t \uparrow c)$. Then from $\mathcal{E}, E; B \vdash t$ via (BINDING-VALUE) and (OWN-DEPENDENT) it follows that $\mathcal{E}, E; B \vdash x : c\langle \sigma \rangle$. Using the rule (IN-BIND3), obtain $\mathcal{E}, E; B \vdash x^{c.j} = k_j$;

Case (p_i is actual)**.** The proof is by identity;

Case (p_i is concrete)**.** p_i is either an ownership parameter or a term variable. These cases are handled by the rules (IN-BIND1) and (IN-BIND2) respectively;

Case ($p_i = ?$)**.** The proof of equality is trivial since then $p_i = q_i = ?$.

□

The following Lemmas C.2.8 and C.2.9 establish the relation between the types of fields and methods in the types related by subtyping, in terms of owner substitutions.

Lemma C.2.8. *If $s \leq s'$ for some $s = c\langle\sigma\rangle$ and $s' = d\langle\sigma'\rangle$ and $f \in dom(\mathcal{F}_c) \cap dom(\mathcal{F}_d)$ then $\sigma(\mathcal{F}_c(f)) = \sigma'(\mathcal{F}_d(f))$.*

Proof. Straightforward by the definition of the subtyping relation (Figure 8.4) and the rule (CLASS) (Figure 8.6). □

Lemma C.2.9. *If $s \leq s'$ for some $s = c\langle\sigma\rangle$ and $s' = d\langle\sigma'\rangle$ and $m \in dom(\mathcal{M}\,\mathcal{T}_c) \cap dom(\mathcal{M}\,\mathcal{T}_d)$ then $\sigma(\mathcal{M}\,\mathcal{T}_c(m)) = \sigma'(\mathcal{M}\,\mathcal{T}_d(m))$.*

Proof. Straightforward by the definition of the subtyping relation (Figure 8.4) and the rule (CLASS) (Figure 8.6). □

Lemma C.2.10 related types assigned to variables and their corresponding runtime values' types in the presence of the binding environment and a heap.

Lemma C.2.10. *If $B(y) = \iota$, $\mathcal{E}, E; B \vdash y : s'$, $\mathcal{E} \vdash H$, $H(\iota) = \langle s, \ldots\rangle$ then $\mathcal{E}, E; B \vdash s \leq s'$*

Proof. The statement of the lemma is an inversion of the rule (BINDING-VALUE) and applying the equivalence due to the rule (IN-BIND3), since $y = \iota \in B$ (Figure 8.11). □

Lemma C.2.11 characterizes the behaviour of the helper function fill, defining the way it replaces unknown owners "?" by dependent owners, so the "residuals" of the type are equivalent

Lemma C.2.11. *If $\mathcal{E}, E; B \vdash c\langle\sigma\rangle$, $x \notin dom(E) \cup dom(B)$ and $c\langle\sigma'\rangle = \mathsf{fill}(x,t)$ then*

$$\mathcal{E}, E; B \vdash c\langle\sigma' \uplus \{x^{c.j} \mapsto u_j\}\rangle = c\langle\sigma \uplus \{?_j \mapsto u_j\}\rangle$$

for some $\mathcal{E}, E; B \vdash u_j$, where $?_j$ is j-th component of the substitution σ (an unknown owner).

Proof. By the definition of fill. □

Lemma C.2.12 relates the relations $\lhd$ and *subtype* via the helper relation $t \uparrow c$.

Lemma C.2.12. *If* $\mathcal{E}, E; B \vdash s \lhd c\langle\sigma\rangle$ *then* $\mathcal{E}, E; B \vdash s \leq c\langle\sigma \uplus \{?_j \mapsto \mathsf{owner}_j(s \uparrow c)\}\rangle$, *where* $?_j$ *is j-th component of the substitution* σ *(an unknown owner).*

Proof. By the definitions of $\lhd$ and $\uparrow$. □

In order to prove the type safety, we need some machinery to deal with the relation $\prec$ and handle it during managing type substitutions, so the rule (G-TYPE) would be respected. The following concept is essential for the further proofs.

Definition C.2.13 (Monotonic owner substitution)**.** An owner substitution $\sigma = \{\alpha_i \mapsto r_i\}_{i \in 1..n}$ is *monotonic* in the typing environment E if $E \vdash \alpha_i \prec \alpha_j \;\Rightarrow\; E \vdash \sigma(\alpha_i) \prec \sigma(\alpha_j)$

Lemma C.2.14 (Composition is monotonic)**.** *If* σ_1 *and* σ_2 *are both* monotonic *in a* well-formed E *and* $im(\sigma_1) \subseteq dom(\sigma_2)$ *then* $\sigma_2 \circ \sigma_1$ *is also monotonic in E.*

Proof. The proof is straightforward by contradiction and the transitivity of $\prec$. □

Lemma C.2.15 (Type comparison and substitutions)**.** *If* $E; B \vdash s \lhd t$*, all* unequivocal *owner constituents of t are* actual *and* σ *is a* monotonic *substitution in E then* $E; B \vdash \sigma(s) \lhd \sigma(t)$.

Proof. The proof is by contradiction and the definition of $\lhd$ since all components of s and t involved into $\lhd$ computations are affected by σ simultaneously and the typing environment E is *well-formed*. □

Finally, we have all ingredients to prove the subject reduction theorem.

Theorem C.2.16 (Subject reduction in $\mathsf{JO}^+_?$ (**Theorem 8.4.6**))**.** *If* $e \in$ **Expr** *in* $\mathsf{JO}^+_?$, $\mathcal{S} = \langle H, B, e, K\rangle$, $\mathcal{E}; \overline{E} \Vdash \mathcal{S}$ *for some well-formed* $\mathcal{E}; \overline{E}$ *and* $\mathcal{S} \Rightarrow \mathcal{S}'$ *then* $\mathcal{E}'; \overline{E}' \Vdash \mathcal{S}'$ *for some well-formed* $\mathcal{E}', \overline{E}'$ *such that* $\mathcal{E}' \gg \mathcal{E}$ *and* $\overline{E} \twoheadrightarrow \overline{E}'$.

Proof. By case analysis on the transition rules (Figures 8.10).

Case (E-LKP)**.** Assume

$$\langle H, B, \mathtt{let}\ x = y.f\ \mathtt{in}\ e, K\rangle \Rightarrow \langle H, B[x \mapsto v], e, K\rangle.$$

1. By (T-STATE) and (TC-MT) or (TC-CALL) depending on K,

   (a) $\mathcal{E} \vdash H$
   (b) $\mathcal{E}, E_0; B \vdash^{c}_{\mathcal{B}} \texttt{let}\ x = y.f\ \texttt{in}\ e : s$ for some unequivocal type s

2. By (T-LET) and (1:b),
   (a) $\mathcal{E}, E_0; B \vdash^{c}_{\mathcal{B}} y.f : t$ for some type t
   (b) $\mathcal{E}, E_0, x : \mathsf{fill}(x,t); B \vdash^{c}_{\mathcal{B}} e : s$

3. By (T-LKP) and (2:a),
   (a) $\mathcal{E}, E_0; B \vdash^{c}_{\mathcal{B}} y : c\langle\sigma\rangle$
   (b) $\mathcal{F}_c(f) = t'$
   (c) $\sigma_y(t') = t$

4. Assume $B(y) = \iota$, otherwise the rest of the proof is trivial. It follows from (1:a), (3:a) and Lemma C.2.10 that
   (a) $H(\iota) = \langle d\langle\sigma'\rangle, f \mapsto v_{f \in \mathrm{dom}(\mathcal{F}_d)}\rangle$
   (b) $\mathcal{E}, E_0; B \vdash d\langle\sigma'\rangle \leq c\langle\sigma\rangle$

5. By Lemma C.2.8, obtain $\mathcal{E}, E_0; B \vdash \sigma(\mathcal{F}_c(f)) = \sigma'(\mathcal{F}_d(f))$

6. By (4), (5) and (IN-BIND2), obtain $\mathcal{E}, E_0; B \vdash \sigma'_\iota(\mathcal{F}_d(f)) = \sigma_y(\mathcal{F}_c(f)) = t$

7. By (1:a), (HEAP) and (HEAP-OBJECT), obtain
   (a) $\mathcal{E}; \emptyset \vdash v_f : s'$
   (b) $\mathcal{E}; \emptyset \vdash s' \lhd \sigma'_\iota(\mathcal{F}_d(f))$

8. By (7:a) and (6) obtain $\mathcal{E}, E_0; B \vdash s' \lhd t$

9. By Lemma C.2.11 obtain $\mathcal{E}, E_0; B \vdash s' \leq s''$ where $c'\langle\sigma''\rangle = \mathsf{fill}(x,t)$ and $s'' = c'\langle\sigma'' \uplus \{x^{c'.j} \mapsto \mathsf{owner}_j(s' \uparrow c')\}\rangle$.

10. Applying (BINDING-VALUE) to (9) and assuming $B' = B, x = v_f$ and $E'_0 = E_0, x : \mathsf{fill}(x,t)$, obtain
    (a) $\mathcal{E}, E'_0; B' \vdash \diamond$
    (b) $\overline{E} \twoheadrightarrow \overline{E}' = E'_0 :: \mathsf{tail}(\overline{E})$
    (c) $\mathcal{E}; \overline{E}'; B' \Vdash \langle e, K\rangle$

11. By (10) via (T-STATE), obtain $\mathcal{E}; \overline{E}' \Vdash \langle H, B[x \mapsto v], e, K\rangle$, which concludes the proof for this case.

Case (E-NEW)**.** Assume

$$\langle H, B, \texttt{let}\ x = \texttt{new}\ c\langle r_{i\in 1..n}\rangle\ \texttt{in}\ e, K\rangle \Rightarrow \langle H', B[x \mapsto \iota], e, K\rangle,$$

where ι is some fresh address in H and $H' \equiv H \uplus \iota \mapsto \langle c\langle B(r_{i\in 1..n})\rangle, f \mapsto \texttt{null}_{f\in \mathrm{dom}(\mathcal{F}_c)}\rangle$

1. By (T-STATE) and (TC-MT) or (TC-CALL) depending on K,
   (a) $\mathcal{E} \vdash H$
   (b) $\mathcal{E}, E_0; B \vdash^{C}_{\mathcal{B}} \texttt{let}\ x = \texttt{new}\ c\langle r_{i\in 1..n}\rangle\ \texttt{in}\ e : s$ for some unequivocal type s
2. By (T-LET) and (1:b), since $\mathsf{fill}(x, c\langle r_{i\in 1..n}\rangle) \equiv c\langle r_{i\in 1..n}\rangle$
   (a) $\mathcal{E}, E_0; B \vdash^{C}_{\mathcal{B}} \texttt{new}\ c\langle r_{i\in 1..n}\rangle : c\langle r_{i\in 1..n}\rangle$
   (b) $\mathcal{E}, E_0, x : c\langle r_{i\in 1..n}\rangle; B \vdash^{C}_{\mathcal{B}} e : s$
3. By (T-NEW), $\mathcal{E}, E_0; B \vdash c\langle r_{i\in 1..n}\rangle$, which implies $\mathcal{E}, E_0; B \vdash r_1 \prec r_i$ for all $i \in 1..n$ by (G-TYPE).
4. By assumption, $r_i \in \mathrm{dom}(B)$ for all $i \in 1..n$, we also have
   (a) $\mathcal{E}, E_0; B \vdash \diamond$, and therefore
   (b) $\mathcal{E}, E_0; B \vdash B(r_i) = r_i \quad \forall i \in 1..n$
   (c) $\mathcal{E}, E_0; B \vdash c\langle B(r_{i\in 1..n})\rangle$
5. Assume
   $$\mathcal{E}' = \mathcal{E}, (\iota : c\langle B(r_{i\in 1..n})\rangle), (\iota \prec B(r_1)).$$
   By Definition 8.1.1, the rule (T-NULL)
   (a) $\widehat{H'} \Rightarrow \mathcal{E}'$
   (b) $\mathcal{E}'; \emptyset \vdash v_f : s_f \quad \mathcal{E}' \vdash s \lhd \sigma_\iota(\mathcal{F}_c(f))$ for some type s_f for all $f \in \mathrm{dom}(\mathcal{F}_c)$
6. By (HEAP-OBJECT), (HEAP), (4:c), (5), obtain
   (a) $\mathcal{E}' \vdash H'$
7. As $\mathcal{E}'$ was defined, obtain
   (a) $\mathcal{E}', E_0, x : c\langle r_{i\in 1..n}\rangle; B \vdash \iota : c\langle r_{i\in 1..n}\rangle$
   (b) $\mathcal{E}', E_0, x : c\langle r_{i\in 1..n}\rangle; B \vdash x : c\langle r_{i\in 1..n}\rangle$
   (c) $x \notin \mathrm{dom}(B)$
8. From (7:a), (7:b) and (7:c), via (BINDING-VALUE), conclude $\mathcal{E}', E'_0; B' \vdash \diamond$, where $E'_0 = E_0, x : c\langle r_{i\in 1..n}\rangle$ and $B' = B[x \mapsto \iota]$.

9. From (8), (6:a) and (1:b), via (T-STATE), conclude
   (a) $\mathcal{E}', E_0' :: \mathsf{tail}(\overline{E}) \Vdash \langle H', B', e, K\rangle$
   (b) $\overline{E} \twoheadrightarrow E_0' :: \mathsf{tail}(\overline{E})$
   (c) $\mathcal{E}' \gg \mathcal{E}$
   (d) $\mathcal{E}'$ is well-formed since $\mathcal{E}$ is well-formed.
10. By (9) via (T-STATE), obtain $\mathcal{E}'; \overline{E}' \Vdash \langle H', B[x \mapsto \iota], e, K\rangle$, which concludes the proof for this case.

Case (E-UPD)**.** Assume $\langle H, B, \mathtt{let}\ x = (y.f = y')\ \mathtt{in}\ e, K\rangle \Rightarrow \langle H', B[x \mapsto v], e, K\rangle$ where $B(y) = \iota$, $B(y') = v$, $o = H(\iota) = \langle d\langle\sigma'\rangle, f \mapsto v_{f \in \mathrm{dom}(\mathcal{F}_c)}\rangle$ and $H' \equiv H \uplus \iota \mapsto o[f \mapsto v]$. We consider $v \neq \mathtt{null}$, otherwise the proof is trivial vial the rule (T-NULL).

1. By (T-STATE) and (TC-MT) or (TC-CALL) depending on K,
   (a) $\mathcal{E} \vdash H$
   (b) $\mathcal{E}, E_0; B \vdash^{C}_{\mathcal{B}} \mathtt{let}\ x = (y.f = y')\ \mathtt{in}\ e : s$ for some unequivocal type s
2. By (T-LET) and (1:b),
   (a) $\mathcal{E}, E_0; B \vdash^{C}_{\mathcal{B}} y.f = y' : t$ for some type t
   (b) $\mathcal{E}, E_0, x : \mathsf{fill}(x,t); B \vdash^{C}_{\mathcal{B}} e : s$
3. By (2:a) and (T-UPD")
   (a) $\mathcal{E}, E_0; B \vdash y : c\langle\sigma\rangle$
   (b) $\sigma_y(\mathcal{F}_c(f)) = t$
   (c) $\mathcal{E}, E_0; B \vdash y' : s'$
   (d) $\mathcal{E}, E_0; B \vdash s' \lhd t$
4. By (3:a) and the assumptions of the case via Lemma C.2.10, obtain $\mathcal{E}, E_0; B \vdash d\langle\sigma'\rangle \leq c\langle\sigma\rangle$
5. By Lemma C.2.8, obtain $\mathcal{E}, E_0; B \vdash \sigma(\mathcal{F}_c(f)) = \sigma'(\mathcal{F}_d(f))$
6. By (4), (5) and (IN-BIND2), obtain $\mathcal{E}, E_0; B \vdash \sigma'_\iota(\mathcal{F}_d(f)) = \sigma_y(\mathcal{F}_c(f)) = t$
7. Considering $B(y') = v$ via (BINDING-VALUE), obtain
   (a) $\mathcal{E} \vdash v : s''$
   (b) $\mathcal{E}, E_0; B \vdash s'' \leq s'$ since $y' = v \in B$
8. By (7:b), (3:d), obtain $\mathcal{E}, E_0; B \vdash s'' \lhd t$ and $t = \sigma'_\iota(\mathcal{F}_d(f))$.

9. Via (7:a), (8), (HEAP) and (HEAP-OBJECT), obtain $\mathcal{E} \vdash s'' \lhd \sigma'_\iota(\mathcal{F}_d(f))$, therefore $\mathcal{E} \vdash H'$
10. By (9) ands Lemma C.2.11 obtain $\mathcal{E}, E_0; B \vdash s'' \leq s'''$ where $c'\langle\sigma''\rangle = \mathsf{fill}(x,t)$ and $s''' = c'\langle\sigma'' \uplus \{x^{c'.j} \mapsto \mathsf{owner}_j(s'' \uparrow c')\}\rangle$.
11. Applying (BINDING-VALUE) to (10) and assuming $B' = B, x = v$ and $E'_0 = E_0, x : \mathsf{fill}(x,t)$, obtain
    (a) $\mathcal{E}; E'_0; B' \vdash \diamond$
    (b) $\overline{E} \twoheadrightarrow \overline{E}' = E'_0 :: \mathsf{tail}(\overline{E})$
    (c) $\mathcal{E}; \overline{E}'; B' \Vdash \langle e, K\rangle$
12. By (11) via (T-STATE), obtain $\mathcal{E}; \overline{E}' \Vdash \langle H', B[x \mapsto v], e, K\rangle$, which concludes the proof for this case.

Case (E-CALL)**.** Assume

$$\langle H, B, \texttt{let}\ x : (t,\sigma) = y.m(y')\ \texttt{in}\ e, K\rangle \Rightarrow \langle H, B', e', \mathbf{call}(x : (t,\sigma), e, B, K)\rangle.$$

1. By assumptions of the case, obtain
    (a) $B(y) = \iota$
    (b) $B(y') = v$
    (c) $H(\iota) = \langle d\langle\sigma'\rangle, \ldots\rangle$
    (d) $\mathcal{M}_d(m) = (x', e', \sigma^*)$ where $\sigma^* = \{\alpha_i \mapsto r_{i\in 1..n}\}$
    (e) $B' = \{\alpha_i \mapsto \sigma^\#(\alpha_i)_{i\in 1..n}, \texttt{this} \mapsto \iota, x' \mapsto v\}$ where $\sigma^\# = (\sigma^* \circ \sigma')$
2. By (T-STATE) and (TC-MT) or (TC-CALL) depending on K,
    (a) $\mathcal{E} \vdash H$
    (b) $\mathcal{E}, E_0; B \vdash^{\mathcal{C}}_{\mathcal{B}} \texttt{let}\ x : (t,\sigma) = y.m(y')\ \texttt{in}\ e : s$ for some unequivocal type s
3. By (T-LET), (T-CALL') and (1:b),
    (a) $\mathcal{E}, E_0; B \vdash y : c\langle\sigma''\rangle$
    (b) $\mathcal{E}, E_0; B \vdash y' : s'$
    (c) $\mathcal{M}\,\mathcal{T}_c(m) = (x', t'' \to t')$
    (d) $\mathcal{E}, E_0; B \vdash s' \lhd \sigma''_y(t'')$
    (e) $\sigma''' \equiv \sigma'' \uplus \{x' \mapsto y'\}$
    (f) $\mathcal{E}, E_0; B \vdash^{\mathcal{C}}_{\mathcal{B}} y.m(y') : \sigma'''_y(t')$ for some type t'

(g) $\mathcal{E}, E_0, x : \mathsf{fill}(x, \sigma'''_y(t')); B \vdash^{C}_{\mathcal{B}} e : s$

4. Put $t \equiv t'$ and $\sigma = \sigma'''_y$, where t, σ are from the assumptions of the case.

5. Without loss of generality, assume that the method m is defined in the body of the class c. Otherwise the reasoning is by induction of the height of method lookup and Lemma C.2.14.

6. By the rules (T-METHOD') and (CLASS) for method m, obtain

   (a) $E' \vdash^{C}_{\mathcal{B}} e' : s''$

   (b) $E \vdash s'' \lhd t'$

   where $E \equiv \mathtt{this} : c\langle \alpha_{i\in 1..n}\rangle, \alpha_1 \prec \mathtt{world}, (\alpha_1 \prec \alpha_i)_{i\in 2..n}$, $E' \equiv E, x' : \mathsf{fill}(x', t'')$ and $\alpha_{i\in 1..n}$ are owner parameters of the class c. By (5) we assume these are same α_i as in (1:e). It is important to notice, that σ is *monotonic* in $\mathcal{E}, E_0, E; B$ (in the assumption that $\mathrm{dom}(E_0) \cap \mathrm{dom}(E') = \emptyset$, which can be achieved by renaming).

7. By (3:a), (1:a), (1:c), (2:a) via Lemma C.2.10, obtain $\mathcal{E}, E_0; B \vdash d\langle \sigma' \rangle \leq c\langle \sigma'' \rangle$.

8. By Lemma C.2.9, obtain $\mathcal{E}, E_0; B \vdash \sigma''(\mathcal{M}\,\mathcal{T}_c(m)) = \sigma'(\mathcal{M}\,\mathcal{T}_d(m))$

9. By assumption (5), obtain $\mathcal{E}, E_0; B \vdash c\langle \sigma'' \rangle = c\langle \sigma^{\#} \rangle$

10. Considering $B(y') = v$ via (BINDING-VALUE), obtain

    (a) $\mathcal{E} \vdash v : s'''$

    (b) $\mathcal{E}, E_0; B \vdash s''' \leq s'$ since $y' = v \in B$

11. By (10:b), (3:d), obtain $\mathcal{E}, E_0; B \vdash s''' \lhd \sigma''_{\mathrm{t}}(t'')$.

12. It follows from the rule (CLASS), (1:d), (1:e), (2:a) via Lemma C.2.14 that $\sigma^{\#}$ is *monotonic* in $\mathcal{E}, E', \tilde{E}$, where $\tilde{E}$ is an environment from the rule (CLASS) for the class d. Considering $\sigma^{\#}$ as a black-box mapping (i.e., excluding the intermediate mapping when defining $\sigma^{\#}$ as a composition), we may drop $\tilde{E}$ and obtain $\sigma^{\#}$ *monotonic* in $\mathcal{E}, E'$.

13. One can see, that the environment E' is well-formed. Our goal now is to show that $\mathcal{E}, E'; B' \vdash \diamond$. We will do it by adding bindings inductively starting from the empty binding environment via ad-hoc"updating" operation $(:=)$, until we obtain required B'

    (a) $\mathcal{E}, E'; \{B_{\mathrm{new}} := \emptyset\} \vdash \diamond$ by (BINDING-EMPTY)

    (b) $\mathcal{E}, E'; \{B_{\mathrm{new}} := B_{\mathrm{old}}, (\alpha_1 = \sigma^{\#}(\alpha_1))\} \vdash \diamond$ by (BINDING-OWNER) and monotonicity of $\sigma^{\#}$ in $\mathcal{E}, E'$

(c) $\mathcal{E}, E'; \{B_{\text{new}} := B_{\text{old}}, (\alpha_i = \sigma^{\#}(\alpha_i)_{i \in 2..n})\} \vdash \diamond$ by (BINDING-OWNER) and monotonicity of $\sigma^{\#}$ in $\mathcal{E}, E'$

(d) $\mathcal{E}, E'; \{B_{\text{new}} := B_{\text{old}}, (\texttt{this} = \iota)\} \vdash \diamond$ by (BINDING-VALUE), since $\mathcal{E} \vdash \iota : d\langle\sigma'\rangle$, by (7) and (8) $\mathcal{E}; B_{\text{old}} \vdash d\langle\sigma'\rangle \leq c\langle\sigma^{\#}\rangle$, $\mathcal{E}; B_{\text{old}} \vdash c\langle\alpha_{1 \in 1..n}\rangle = c\langle\sigma^{\#}\rangle$ and all components of codomain of σ' and $\sigma^{\#}$ are actual, i.e., their equalities do not depend on local environments.

(e) $\mathcal{E}, E'; \{B_{\text{new}} := B_{\text{old}}, (x' = v)\} \vdash \diamond$ by (9), (11) and (BINDING-VALUE) using Lemma C.2.11.

14. Finally, we obtain

(a) $\mathcal{E}; E'; B' \vdash \diamond$ by (13)

(b) $\mathcal{E}; E'; B' \vdash^{c}_{\mathcal{B}} e' : s''$ by (6:a)

(c) $\mathcal{E}; E'; B' \vdash s'' \lhd t$ by (6:b) and (4)

(d) $(\mathcal{E}, E_0, B \vdash \sigma(r) = k) \Leftrightarrow (\mathcal{E}, E', B' \vdash r = k) \ \ \forall r \in \text{dom}(\sigma)$ by construction of σ

(e) $\mathcal{E}, E_0, x : \mathsf{fill}(x, \sigma(t)); B \Vdash \langle e, K\rangle$ by (3:g) and (4)

15. From (14) via (TC-CALL) conclude

(a) $\overline{E} \twoheadrightarrow \overline{E}' = E' :: \overline{E}$

(b) $\mathcal{E}; \overline{E}'; B' \Vdash \langle e', \mathbf{call}(x : (t, \sigma, ,)e, B, K)\rangle$

16. From (15) via (T-STATE), obtain $\mathcal{E}; \overline{E}' \Vdash \langle H, B', e', \mathbf{call}(x : (t, \sigma), e, B, K)\rangle$, which concludes the proof for this case.

Case (E-RETURN)**.** Assume

$$\langle H, B, y, \mathbf{call}(x : (t, \sigma), e, B', K)\rangle \Rightarrow \langle H, B'[x \mapsto B(y)], e, K\rangle.$$

1. By the rules (T-STATE) and (TC-CALL) for the assumptions, obtain:

(a) $\mathcal{E} \vdash H$

(b) $\mathcal{E}, E_0; B \vdash^{c}_{\mathcal{B}} y : s$

(c) $\mathcal{E}, E_0; B \vdash s \lhd t$

(d) $\mathcal{E}, E_1, (x : \mathsf{fill}(x, \sigma(t))); \overline{E}; B' \Vdash \langle e', K\rangle$

2. Let us consider the case when $b(y) = v \neq \texttt{null}$, otherwise the proof is trivial. By (1:a) and (1:b), obtain

(a) $\mathcal{E} \vdash v : s'$

(b) $\mathcal{E}, E_0; B \vdash s' \leq s$

3. Because of the presence of B in the premise of (2:b) there are different representations of the type s' thanks to equalities provided by the rules (IN-BIND*). Let us chose s'' such that

   (a) $\mathcal{E}, E_0; B \vdash s'' = s'$

   (b) $\forall r_i \in \mathsf{owners}(s'').\ r_i = \begin{cases} r' & \text{if } r' \in \mathrm{dom}(\sigma) \\ k \quad \text{for some } \mathsf{actual} \text{ owner } k & \text{otherwise} \end{cases}$

   (c) We have $\mathcal{E}, E_0; B \vdash s'' \lhd t$ as a corollary of (3:a), (3:b) and (1:c)

   In words, all owner constituents of s'' are either transformed by σ or are actual.

4. Without loss of generality, assume that $\mathrm{dom}(E_0) \cap \mathrm{dom}(E_1) = \emptyset$ (this can be reached by α-renaming). This implies, according to well-formedness of typing-environment pairs, that $\mathrm{dom}(B) \cap \mathrm{dom}(B') = \emptyset$ as well. One can see that $\mathcal{E}, E_0, E_1; B, B' \vdash s'' \lhd t$.

5. The monotonicity of σ has been shown in the case (E-CALL), step 6. By Lemma C.2.15 and (4), obtain $\mathcal{E}, E_0, E_1; B, B' \vdash \sigma(s'') \lhd \sigma(t)$.

6. Since no more $r_i \in \mathrm{dom}(\sigma)$ is mentioned in the statement from (5), we can drop E_0 and B from the premise of the statement: $\mathcal{E}, E_1; B' \vdash \sigma(s'') \lhd \sigma(t)$.

7. Applying the property of σ that $(\mathcal{E}, E_0, B \vdash r = k) \Leftrightarrow (\mathcal{E}, E_1, B' \vdash \sigma(r) = k)\ \forall r \in \mathrm{dom}(\sigma)$ (from (TC-CALL)), we can "switch" back from $\sigma(s'')$ to s' (recall, that all ownership arguments of s' are actual, i.e., some k).

8. By (7), obtain $\mathcal{E}, E_1; B' \vdash s' \lhd \sigma(t)$

9. By (8) and Lemma C.2.11 obtain $\mathcal{E}, E_1; B' \vdash s' \leq s'''$ where $c'\langle\sigma'\rangle = \mathsf{fill}(x, \sigma(t))$ and $s''' = c'\langle\sigma' \uplus \{x^{c'.j} \mapsto \mathsf{owner}_j(s' \uparrow c')\}\rangle$.

10. Applying (BINDING-VALUE) to (9) and assuming $B'' = B, x = v$ and $E_1' = E_1, x : \mathsf{fill}(x, \sigma(t))$, obtain

    (a) $\mathcal{E}, E_1'; B'' \vdash \diamond$

    (b) $E_0 :: E_1 :: \overline{E} \twoheadrightarrow \overline{E}' = E_1' :: \overline{E}$

    (c) $\mathcal{E}; \overline{E}'; B'' \Vdash \langle e, K\rangle$

11. From (10) via (T-STATE), obtain $\mathcal{E}; \overline{E}' \Vdash \langle H, B'', e, K\rangle$, which concludes the proof for this case.

Case (E-CAST1)**.** Assume $\langle H, B, \mathtt{let}\ x = \langle t\rangle y\ \mathtt{in}\ e, K\rangle \Rightarrow \langle H, B[x \mapsto B(y)], e, K\rangle$ where $B(y) = v$. We consider $v \neq \mathtt{null}$, otherwise the proof is trivial vial the rule (T-NULL).

1. By (T-STATE) and (TC-MT) or (TC-CALL) depending on K,
   (a) $\mathcal{E} \vdash H$
   (b) $\mathcal{E}, E_0; B \vdash^{C}_{\mathcal{B}} \texttt{let}\ x = \langle t \rangle y\ \texttt{in}\ e : s$ for some unequivocal type s
2. By (T-LET) and (1:b),
   (a) $\mathcal{E}, E_0; B \vdash^{C}_{\mathcal{B}} \langle t \rangle y : t'$ for some type t'
   (b) $\mathcal{E}, E_0, x : \mathsf{fill}(x,t); B \vdash^{C}_{\mathcal{B}} e : s$
3. By (T-CAST) and (2:a),
   (a) $\mathcal{E}, E_0; B \vdash y : s'$
   (b) $\mathcal{E}, E_0; B \vdash t$
   (c) $\mathcal{E}, E_0; B \vdash s' \lesssim t$
   (d) $\mathcal{E}, E_0; B \vdash t' = t$
4. By assumptions we have $H; B \vdash \mathsf{cast}(t, y)$ that implies the following statements via the rule (CAST-CHECK):
   (a) $H; B \vdash t \ltimes t''$
   (b) $H(v) = \langle s, \ldots \rangle$ for some $s'' = d\langle \sigma' \rangle$
   (c) $\widehat{H} \vdash s'' \lhd t''$
5. By (1:a), (3:b), (4:a) and Lemma C.2.7, obtain
   (a) $\mathcal{E}, E_0; B \vdash t = t''$
   (b) $\mathcal{E}, E_0; B \vdash s'' \lhd t$
6. By (5:b) ands Lemma C.2.11 obtain $\mathcal{E}, E_0; B \vdash s'' \leq s'''$ where $c'\langle \sigma'' \rangle = \mathsf{fill}(x,t)$ and $s''' = c'\langle \sigma'' \uplus \{x^{c'.j} \mapsto \mathsf{owner}_j(s'' \uparrow c')\} \rangle$.
7. Applying (BINDING-VALUE) to (6) and assuming $B' = B, x = v$ and $E'_0 = E_0, x : \mathsf{fill}(x,t)$, obtain
   (a) $\mathcal{E}; E'_0; B' \vdash \diamond$
   (b) $\overline{E} \twoheadrightarrow \overline{E}' = E'_0 :: \mathsf{tail}(\overline{E})$
   (c) $\mathcal{E}; \overline{E}'; B' \Vdash \langle e, K \rangle$
8. From (7) via (T-STATE), obtain $\mathcal{E}; \overline{E}' \Vdash \langle H, B[x \mapsto B(y)], e, K \rangle$, which concludes the proof for this case.

Case (E-CAST2)**.** The proof is trivial by the rules (T-STATE) and (TC-FAIL).

Case (E-BOUNDARY1)**.** The proof is similar to the case (E-UPD).

Case (E-BOUNDARY2)**.** The proof is trivial by the rules (T-STATE) and (TC-FAIL).

Case (E-FINAL)**.** Assume $\langle H, B, y, \mathbf{mt}\rangle \Rightarrow \langle H, B, B(y), \mathbf{mt}\rangle$. The proof of this case follows immediately from the rules (T-STATE), (TC-MT) and the rule (BINDING-VALUE) considering that $\mathcal{E}, E :: \mathbf{nil}; B \vdash \diamond$ for some environments $\mathcal{E}$ and E.

□

Now, we are ready to prove the last crucial theorem about the calculus of gradual ownership types: the OAD preservation in $\mathsf{JO}^{+}_{?}$.

Theorem C.2.17 (OAD preservation in $\mathsf{JO}^{+}_{?}$ (**Theorem 8.4.7**))**.** *If* $e \in$ **Expr** *in* $\mathsf{JO}^{+}_{?}$*,* $\mathcal{S} = \langle H, B, e, K\rangle$*,* $\mathcal{E}; \overline{E} \Vdash \mathcal{S}$*,* $\mathsf{OAD}(H)$ *and* $\mathcal{S} \Rightarrow \mathcal{S}'$ *for some* $\mathcal{S}' = \langle H', \_, \_, \_\rangle$ *then* $\mathsf{OAD}(H')$.

Proof. By case analysis on the transition relation $\mathcal{S} \Rightarrow \mathcal{S}'$. In fact, there are only three transitions in the operational semantics of $\mathsf{JO}^{+}_{?}$ that can affect the OAD invariant: (E-NEW), (E-UPD), and (E-BOUNDARY1). We consider all of them separately.

Case (E-NEW)**.** According to the transition rule, $H' \equiv H \uplus \iota \mapsto o$. All fields of the newly allocated object o point to `null`. No fields of any object point to ι so far. So we have $\mathsf{OAD}(H')$.

Case (E-UPD)**.** By the assumption of the theorem, we have

$$\mathcal{E}; \overline{E} \Vdash \langle H, B, \texttt{let } z = (x.f = y) \texttt{ in } e, K\rangle.$$

Assume $B(x) \neq \texttt{null}$ and $B(y) \neq \texttt{null}$, otherwise the proof is trivial.

1. By (T-STATE) and then (TC-CALL) or (TC-MT) depending on k for the hypothesis, by premises we obtain via (T-LET).
   (a) $\mathcal{E}, E_0; B \vdash^{c}_{\mathcal{B}} x.f = y : t$ for some type t
   (b) $\mathcal{E} \vdash H$
2. By (T-UPD") for (1:a) and the premises, obtain
   (a) $\mathcal{E}, E_0; B \vdash x : c\langle\sigma\rangle$
   (b) $\mathcal{F}_c(f) = t'$
   (c) $\mathcal{E}, E_0; B \vdash y : s$
   (d) $\mathcal{E}, E_0; B \vdash s \lhd \sigma_x(t')$

(e) $\mathcal{E}, E_0; B \vdash \textbf{specified}(\sigma_x(t'))$

(f) $t = \sigma_x(t')$

3. From (2:d) and (2:e) via Definition 8.2.5 obtain $\mathsf{owner}(s)$ is *unequivocal*.

4. By (2:c), (3) and (BINDING-VALUE), assuming $v = B(y)$ and $o = \mathsf{owner}(H(v))$, obtain $\mathcal{E}, E_0; B \vdash o = \mathsf{owner}(s)$. Moreover, by (2:d), (2:f) and (3), $\mathcal{E}, E_0; B \vdash o = \mathsf{owner}(t)$

5. Inverting subsequently (HEAP) and (HEAP-OBJECT) for (1:b) and (2:a), we obtain

   (a) $\mathcal{E}; \emptyset \vdash c\langle\sigma\rangle$

   (b) $\mathcal{E}; \emptyset \vdash \iota \prec \mathsf{owner}(c\langle\sigma\rangle)$ where $\iota = B(x)$

6. Since all owners in the codomain of σ are actual, we invert the rule (G-TYPE) for (5:a) to obtain

   (a) $\forall k \in \mathsf{owners}(c\langle\sigma\rangle).\ \mathcal{E}; \emptyset \vdash \mathsf{owner}(c\langle\sigma\rangle) \prec k$

7. By (2:a), (2:b), since o is actual via the rule (CLASS) obtain $o \in \mathsf{owners}(c\langle\sigma\rangle)$.

8. From (5:a), (5:b) and (7), conclude $\mathcal{E}; \emptyset \vdash \iota \prec o$.

9. There are no other changes during the step from H to H', so for other fields of the object $H(\iota)$ the proof is by identity.

10. By Definition 8.1.1 of heap flattening $\widehat{H} = \widehat{H'}$.

11. Invert the rule (HEAP) for $\mathcal{E}$ and H' by type preservation ($\mathcal{E} \vdash H'$) via (8), since $H' \Rightarrow \mathcal{E}$, obtain $\widehat{H'}; \emptyset \vdash \iota \prec o$, which matches the definition of the OAD invariant (Definition 8.1.2) and concludes the proof for this case.

Case (E-BOUNDARY1)**.** Invert the rule (BOUNDARY-CHECK) for the updated object referred by x at the heap location ι. It matches exactly the definition of the OAD invariant (Definition 8.1.2).

□

Index

Bibliography

[1] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996. Cited on pages 85 and 115.

[2] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. A functional correspondence between evaluators and abstract machines. In Dale Miller, editor, *PPDP'03: Proceedings of the 5th ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 8–19, August 2003. Cited on pages 13, 28, 72, and 85.

[3] Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. A functional correspondence between call-by-need evaluators and lazy abstract machines. *Information Processing Letters*, 90(5):223–232, 2004. Extended version available as the technical report BRICS-RS-04-03. Cited on pages 28 and 85.

[4] Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. A functional correspondence between monadic evaluators and abstract machines for languages with computational effects. *Theoretical Computer Science*, 342(1):149–172, 2005. Extended version available as the technical report BRICS RS-04-28. Cited on pages 28, 51, 85, and 86.

[5] Amal Ahmed, Robert Bruce Findler, Jeremy G. Siek, and Philip Wadler. Blame for all. In Mooly Sagiv, editor, *POPL '11: Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 201–214, Austin, Texas, USA, 2011. Cited on page 164.

[6] Jonathan Aldrich and Craig Chambers. Ownership Domains: Separating Aliasing Policy from Mechanism. In Martin Odersky, editor, *ECOOP 2004: Proceedings of the 18th European conference on Object-Oriented Programming*, volume 3086 of *Lecture Notes in Computer Science*, pages 1–25, Oslo, Norway, 2004. Springer-Verlag. Cited on pages 148, 153, and 154.

[7] Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. Alias annotations for program understanding. In *OOPSLA '02: Proceedings of the 17th ACM*

SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pages 311–330, Seattle, Washington, USA, 2002. ACM. Cited on page 159.

[8] Davide Ancona. Coinductive big-step operational semantics for type soundness of Java-like languages. In *Proceedings of the 13th Workshop on Formal Techniques for Java-Like Programs (FTfJP '11)*, Lancaster, UK, 2011. ACM. Cited on page 23.

[9] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, Department of Computer Science, University of Copenhagen, Copenhagen, Denmark, May 1994. DIKU Rapport 94/19. Cited on page 159.

[10] Christopher Anderson and Sophia Drossopoulou. BabyJ: From Object Based to Class Based Programming via Types. *ENTCS*, 82(8):53 – 81, 2003. Cited on page 156.

[11] Konrad Anton and Peter Thiemann. Deriving type systems and implementations for coroutines. In Kazunori Ueda, editor, *APLAS 2010, Proceedings of the 8th Asian Symposium on Programming Languages and Systems*, volume 6461 of *Lecture Notes in Computer Science*, pages 63–79, Shanghai, China, Dec 2010. Springer. Cited on page 86.

[12] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 2006. Cited on pages 17 and 35.

[13] Vincent Balat and Olivier Danvy. Memoization in type-directed partial evaluation. In *Proceedings of the 1st ACM SIGPLAN/SIGSOFT conference on Generative Programming and Component Engineering*, GPCE '02, pages 78–92, London, UK, 2002. Springer-Verlag. Cited on page 38.

[14] Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. Design and correctness of program transformations based on control-flow analysis. In Naoki Kobayashi and Benjamin C. Pierce, editors, *Theoretical Aspects of Computer Software, 4th International Symposium, TACS 2001*, volume 2215 of *Lecture Notes in Computer Science*, pages 420–447, Sendai, Japan, October 2001. Springer-Verlag. Cited on pages 40 and 43.

[15] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004. Cited on page 92.

[16] Malgorzata Biernacka, Dariusz Biernacki, and Olivier Danvy. An Operational Foundation for Delimited Continuations in the CPS Hierarchy. *Logical Methods in Computer Science*, 1(2), 2005. Cited on page 85.

[17] Malgorzata Biernacka and Olivier Danvy. A concrete framework for environment machines. *Transactions on Computational Logic*, 9(1):6+, 2007. Cited on pages 13, 55, 85, and 90.

[18] Malgorzata Biernacka and Olivier Danvy. Towards Compatible and Interderivable Semantic Specifications for the Scheme Programming Language, Part II: Reduction Semantics and Abstract Machines. In Jens Palsberg, editor, *Semantics and Algebraic Specification*, volume 5700 of *Lecture Notes in Computer Science*, pages 186–206. Springer, 2009. Cited on page 85.

[19] Dariusz Biernacki. *Operational Aspects of Programming Languages with Delimited Continuations*. PhD thesis, BRICS PhD School, University of Aarhus, Aarhus, Denmark, October 2005. Cited on page 38.

[20] Dariusz Biernacki and Sergueï Lenglet. Applicative bisimulations for delimited-control operators. In Lars Birkedal, editor, *FoSSaCS 2012, Proceedings of the 15th International Conference Foundations of Software Science and Computational Structures*, volume 7213 of *Lecture Notes in Computer Science*, pages 119–134, Tallinn, Estonia, 2012. Springer. Cited on page 38.

[21] Bruno Blanchet. Escape analysis: correctness proof, implementation and experimental results. In Luca Cardelli, editor, *POPL '98: Proceedings of the 25th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 25–37, San Diego, California, January 1998. Cited on page 159.

[22] Ana Bove and Venanzio Capretta. Modelling general recursion in type theory. *Mathematical Structures in Computer Science*, 15(4):671–708, 2005. Cited on page 92.

[23] Chandrasekhar Boyapati, Barbara Liskov, and Liuba Shrira. Ownership types for object encapsulation. In Greg Morrisett, editor, *POPL '03: Proceedings of the 30th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 213–223, New Orleans, Louisiana, 2003. Cited on pages 142 and 148.

[24] Chandrasekhar Boyapati and Martin Rinard. A parameterized type system for race-free Java programs. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 56–69, Tampa, Florida, USA, 2001. ACM. Cited on page 95.

[25] Chandrasekhar Boyapati, Alexandru Salcianu, William Beebee, Jr., and Martin Rinard. Ownership types for safe region-based memory management in real-time Java. In *PLDI '03: Proceedings of the 2003 ACM SIGPLAN conference*

on Programming language design and implementation, pages 324–337, San Diego, CA, USA, June 2003. ACM. Cited on page 95.

[26] Nicholas Cameron and Sophia Drossopoulou. Existential quantification for variant ownership. In Giuseppe Castagna, editor, *ESOP '09: Proceedings of the 18th European Symposium on Programming Languages and Systems*, volume 5502 of *Lecture Notes in Computer Science*, pages 128–142, York, UK, 2009. Springer-Verlag. Cited on pages 148 and 157.

[27] Nicholas R. Cameron, Sophia Drossopoulou, James Noble, and Matthew J. Smith. Multiple ownership. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems, languages and applications*, pages 441–460, Montreal, Quebec, Canada, 2007. ACM. Cited on page 153.

[28] Brian Chin, Shane Markstrum, Todd D. Millstein, and Jens Palsberg. Inference of user-defined type qualifiers and qualifier rules. In Peter Sestoft, editor, *ESOP 2006: Programming Languages and Systems, 15th European Symposium on Programming*, volume 3924 of *Lecture Notes in Computer Science*, pages 264–278, Vienna, Austria, 2006. Springer. Cited on page 158.

[29] Adriana E. Chis, Nick Mitchell, Edith Schonberg, Gary Sevitsky, Patrick O'Sullivan, Trevor Parsons, and John Murphy. Patterns of memory inefficiency. In Mira Mezini, editor, *ECOOP 2011: Proceedings of the 25th European conference on Object-Oriented Programming*, volume 6831 of *Lecture Notes in Computer Science*, pages 383–407, Lancaster, UK, 2011. Springer-Verlag. Cited on page 164.

[30] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for Java. In A. Michael Berman, editor, *OOPSLA '99: Proceedings of the 14th annual ACM SIGPLAN conference on Object-oriented programming systems, languages and applications*, pages 1–19, Denver, Colorado, USA, 1999. ACM. Cited on page 154.

[31] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940. Cited on pages 6 and 31.

[32] Alonzo Church. *The Calculi of Lambda-Conversion*. Princeton University Press, 1941. Cited on pages 21 and 31.

[33] Dave Clarke and Sophia Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 292–310, Seattle, Washington, USA, 2002. ACM. Cited on pages 95, 99, 102, 104, 108, 110, 115, 118, 130, 147, and 164.

[34] Dave Clarke and Ilya Sergey. A semantics for context-oriented programming with layers. In *COP '09: Proceedings of International Workshop on Context-Oriented Programming*, pages 1–6, Genova, Italy, 2009. ACM. Cited on page 24.

[35] Dave Clarke and Tobias Wrigstad. External uniqueness is unique enough. In Luca Cardelli, editor, *ECOOP 2003: Proceedings of the 18th European conference on Object-Oriented Programming*, volume 2743 of *Lecture Notes in Computer Science*, pages 176–200, Darmstadt, Germany, 2003. Springer-Verlag. Cited on pages 151 and 153.

[36] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In Craig Chambers, editor, *OOPSLA '98: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 48–64, Vancouver, British Columbia, Canada, 1998. Cited on pages 110, 118, 130, and 135.

[37] David Gerard Clarke. *Object ownership and containment*. PhD thesis, University of New South Wales, New South Wales, Australia, 2001. Cited on pages 110, 138, 142, 144, 147, and 157.

[38] William D. Clinger. Proper tail recursion and space efficiency. In Keith D. Cooper, editor, *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Languages Design and Implementation*, pages 174–185, Montréal, Canada, June 1998. Cited on page 33.

[39] Patrick Cousot. Types as abstract interpretations. In Neil D. Jones, editor, *POPL '97: Proceedings of the 24th Annual ACM Symposium on Principles of programming languages*, pages 316–331, Paris, France, January 1997. Cited on page 20.

[40] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Ravi Sethi, editor, *POPL '77: Proceedings of the Fourth Annual ACM Symposium on Principles of programming languages*, pages 238–252, Los Angeles, California, January 1977. Cited on pages 29 and 161.

[41] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In Barry K. Rosen, editor, *POPL '79: Proceedings of the Sixth Annual ACM Symposium on Principles of programming languages*, pages 269–282, San Antonio, Texas, January 1979. Cited on page 161.

[42] Patrick Cousot and Radhia Cousot. Higher-order abstract interpretation (and application to comportment analysis generalizing strictness, termination, projection and PER analysis of functional languages), invited paper. In

Henri Bal, editor, *Proceedings of the Fifth IEEE International Conference on Computer Languages*, pages 95–112, Toulouse, France, May 1994. Cited on page 29.

[43] Patrick Cousot and Radhia Cousot. Basic concepts of abstract interpretation. In René Jacquart, editor, *Building the Information Society*, pages 359–366. Kluwer Academic Publishers, 2004. Cited on page 29.

[44] Dave Cunningham, Werner Dietl, Sophia Drossopoulou, Adrian Francalanza, Peter Müller, and Alexander J. Summers. Universe types for topology and encapsulation. In Frank S. Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul Roever, editors, *FMCO '07: 6th International Symposium on Formal Methods for Components and Objects*, pages 72–112. Springer-Verlag, 2008. Cited on pages 159 and 161.

[45] Haskell B. Curry and Robert Feys. *Combinatory Logic, Volume I.* Studies in Logic and the Foundations of Mathematics. North-Holland Publishing Company, 1958. Cited on page 32.

[46] Olivier Danvy. Three steps for the CPS transformation. Technical Report CIS-92-2, Kansas State University, Manhattan, Kansas, December 1991. Cited on pages 39 and 78.

[47] Olivier Danvy. Back to direct style. *Science of Computer Programming*, 22(3):183–195, 1994. Cited on page 39.

[48] Olivier Danvy. Type-directed partial evaluation. In Guy L. Steele Jr., editor, *POPL '96: Proceedings of the 23rd Annual ACM Symposium on Principles of programming languages*, pages 242–257, St. Petersburg Beach, Florida, January 1996. Cited on page 38.

[49] Olivier Danvy. An Analytical Approach to Program as Data Objects. DSc Thesis, Department of Computer Science, Aarhus University, October 2006. Cited on pages 13, 72, and 85.

[50] Olivier Danvy. Refunctionalization at work. In *Preliminary proceedings of the 8th International Conference on Mathematics of Program Construction (MPC '06)*, Kuressaare, Estonia, July 2006. Invited talk. Cited on page 41.

[51] Olivier Danvy. Defunctionalized interpreters for programming languages. In Peter Thiemann, editor, *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP'08)*, pages 131–142, Victoria, British Columbia, Canada, September 2008. Cited on pages 13, 25, 51, and 85.

[52] Olivier Danvy. From Reduction-Based to Reduction-Free Normalization. In Pieter Koopman, Rinus Plasmeijer, and Doaitse Swierstra, editors, *Advanced*

Functional Programming, Sixth International School, number 5382 in Lecture Notes in Computer Science, pages 66–164, Nijmegen, The Netherlands, May 2008. Springer. Lecture notes including 70+ exercises. Cited on pages 13, 14, 51, 55, 58, 60, 85, and 89.

[53] Olivier Danvy and Andrzej Filinski. Abstracting Control. In *In Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, pages 151–160. ACM Press, 1990. Cited on pages 37 and 77.

[54] Olivier Danvy and Andrzej Filinski. Representing control, a study of the CPS transformation. In Mitchell Wand, editor, *Special issue on the 1990 ACM Conference on Lisp and Functional Programming*, Mathematical Structures in Computer Science, Vol. 2, No. 4, pages 361–391. Cambridge University Press, December 1992. Cited on page 39.

[55] Olivier Danvy and John Hatcliff. On the transformation between direct and continuation semantics. In Stephen Brookes, Michael Main, Austin Melton, Michael Mislove, and David Schmidt, editors, *Proceedings of the 9th Conference on Mathematical Foundations of Programming Semantics*, volume 802 of *Lecture Notes in Computer Science*, pages 627–648, New Orleans, Louisiana, April 1993. Springer-Verlag. Cited on page 35.

[56] Olivier Danvy and Jacob Johannsen. Inter-deriving semantic artifacts for object-oriented programming. *Journal of Computer and System Sciences*, 76(5):302–323, 2010. Cited on page 85.

[57] Olivier Danvy and Julia L. Lawall. Back to direct style II: First-class continuations. In William Clinger, editor, *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, LISP Pointers, Vol. V, No. 1, pages 299–310, San Francisco, California, June 1992. Cited on pages 14 and 39.

[58] Olivier Danvy and Kevin Millikin. A Rational Deconstruction of Landin's SECD Machine with the J Operator. *Logical Methods in Computer Science*, 4(4), November 2008. Cited on pages 9, 14, 46, 51, 72, 77, and 85.

[59] Olivier Danvy and Kevin Millikin. On the equivalence between small-step and big-step abstract machines: a simple application of lightweight fusion. *Information Processing Letters*, 106(3):100–109, 2008. Cited on pages 42, 51, 54, and 62.

[60] Olivier Danvy and Kevin Millikin. Refunctionalization at Work. *Science of Computer Programming*, 74(8):534–549, 2009. Cited on pages 14, 51, and 68.

[61] Olivier Danvy, Kevin Millikin, Johan Munk, and Ian Zerny. Defunctionalized Interpreters for Call-by-Need Evaluation. In Matthias Blume, Naoki Kobayashi,

and Germán Vidal, editors, *FLOPS 2010: Proceedings of the 10th International Symposium, Functional and Logic Programming*, volume 6009 of *Lecture Notes in Computer Science*, pages 240–256, Sendai, Japan, Apr 2010. Springer. Cited on pages 24, 86, and 90.

[62] Olivier Danvy and Lasse R. Nielsen. Defunctionalization at work. In Harald Søndergaard, editor, *PPDP'01: Proceedings of the 3rd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 162–174, Firenze, Italy, September 2001. Extended version available as the technical report BRICS RS-01-23. Cited on pages 13, 14, 40, 51, 68, and 85.

[63] Olivier Danvy and Lasse R. Nielsen. Refocusing in reduction semantics. Technical Report BRICS RS-04-26, Department of Computer Science, University of Aarhus, Aarhus, Denmark, November 2004. A preliminary version appears in the informal proceedings of the Second International Workshop on Rule-Based Programming (RULE 2001), Electronic Notes in Theoretical Computer Science, Vol. 59.4. Cited on page 61.

[64] Olivier Danvy and Ulrik P. Schultz. Lambda-dropping: transforming recursive equations into programs with block structure. In A. Michael Berman, editor, *Proceedings of the Sixth ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 90–106, Amsterdam, The Netherlands, December 1997. Cited on page 44.

[65] Olivier Danvy and Ulrik Pagh Schultz. Lambda-lifting in quadratic time. In Zhenjiang Hu and Mario Rodríguez-Artalejo, editors, *FLOPS 2002: Proceedings of the 6th International Symposium on Functional and Logic Programming*, volume 2441 of *Lecture Notes in Computer Science*, pages 134–151, Aizu, Japan, April 2002. Springer. Cited on pages 33 and 43.

[66] Olivier Danvy and Ian Zerny. Three syntactic theories for combinatory graph reduction. In María Alpuente, editor, *LOPSTR 2010: Proceedings of the 20th International Symposium on Logic-Based Program Synthesis and Transformation*, volume 6564 of *Lecture Notes in Computer Science*, pages 3–32, Hagenberg, Austria, Jul 2011. Springer. Cited on page 86.

[67] Zaynah Dargaye and Xavier Leroy. Mechanized verification of CPS transformations. In *LPAR '07: Proceedings of the 14th international conference on Logic for programming, artificial intelligence and reasoning*, pages 211–225, Yerevan, Armenia, 2007. Springer-Verlag. Cited on page 92.

[68] Alain Deutsch. Interprocedural may-alias analysis for pointers: Beyond *k*-limiting. In Vivek Sarkar, editor, *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Languages Design and Implementation*, pages 230–241, Orlando, Florida, June 1994. Cited on page 159.

[69] Werner Dietl, Michael D. Ernst, and Peter Müller. Tunable Static Inference for Generic Universe Types. In Mira Mezini, editor, *ECOOP 2011: Proceedings of the 25th European conference on Object-Oriented Programming*, volume 6831 of *Lecture Notes in Computer Science*, pages 333–357, Lancaster, UK, 2011. Springer-Verlag. Cited on pages 96, 161, and 162.

[70] Werner Dietl and Peter Müller. Runtime universe type inference. In *IWACO '07: International Workshop on Aliasing, Confinement and Ownership in object-oriented programming*, Berlin, Germany, 2007. Cited on page 159.

[71] Paul Downen and Zena M. Ariola. A systematic approach to delimited control with multiple prompts. In Helmut Seidl, editor, *ESOP '12: Proceedings of the 21th European Symposium on Programming Languages and Systems*, volume 7211, pages 234–253, Tallinn, Estonia, 2012. Springer-Verlag. Cited on page 38.

[72] Christopher J. Dutchyn. *Dynamic Join Points: Model and Interactions*. PhD thesis, The University of British Columbia, Vancouver, British Columbia, Canada, November 2006. Cited on page 91.

[73] Torbjörn Ekman and Görel Hedin. Rewritable reference attributed grammars. In Martin Odersky, editor, *ECOOP 2004: Proceedings of the 18th European conference on Object-Oriented Programming*, volume 3086 of *Lecture Notes in Computer Science*, pages 144–169, Oslo, Norway, 2004. Springer-Verlag. Cited on pages 90, 104, and 165.

[74] Torbjörn Ekman and Görel Hedin. The JastAdd extensible Java compiler. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems, languages and applications*, pages 1–18, Montreal, Quebec, Canada, 2007. ACM. Cited on pages 104, 137, 138, and 165.

[75] Chucky Ellison, Traian Florin Şerbănuţă, and Grigore Roşu. A rewriting logic approach to type inference. In *Recent Trends in Algebraic Development Techniques*, volume 5486 of *Lecture Notes in Computer Science*, pages 135–151. Springer, 2009. Revised Selected Papers from the 19th International Workshop on Algebraic Development Techniques (WADT'08). Cited on page 91.

[76] Erik Ernst, Klaus Ostermann, and William R. Cook. A virtual class calculus. In Simon Peyton Jones, editor, *POPL '06: Proceedings of the 33rd annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 270–282, Charleston, South Carolina, January 2006. Cited on page 23.

[77] Manuel Fähndrich and K. Rustan M. Leino. Declaring and checking non-null types in an object-oriented language. In *OOPSLA '03: Proceedings of*

the 18th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pages 302–312, Anaheim, California, USA, 2003. Cited on pages 96 and 135.

[78] Matthias Felleisen. The theory and practice of first-class prompts. In Jeanne Ferrante and P. Mager, editors, *POPL '88: Proceedings of the Fifteenth Annual ACM Symposium on Principles of programming languages*, pages 180–190, San Diego, CA, USA, January 1988. Cited on page 37.

[79] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex, 1st edition*. The MIT Press, August 2009. Cited on pages 9, 11, 25, 26, 33, 55, 91, 127, 169, and 218.

[80] Matthias Felleisen and Daniel P. Friedman. Control operators, the SECD machine, and the λ-calculus. In Martin Wirsing, editor, *Formal Description of Programming Concepts III*, pages 193–217. Elsevier Science Publishers B.V. (North-Holland), Amsterdam, 1986. Cited on pages 11, 28, 68, and 80.

[81] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103(2):235–271, 1992. Cited on page 24.

[82] Darrell Ferguson and Dwight Deugo. Call with Current Continuation Patterns. Technical report, School of Computer Science, Carleton University, 2001. Cited on page 36.

[83] Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In Simon Peyton Jones, editor, *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP'02)*, pages 48–59, Pittsburgh, Pennsylvania, September 2002. Cited on pages 157 and 164.

[84] Adam Fischbach and John Hannan. Specification and correctness of lambda lifting. *Journal of Functional Programming*, 13(3):509–543, 2003. Cited on page 43.

[85] Michael J. Fischer. Lambda-calculus schemata. In Carolyn L. Talcott, editor, *Special issue on continuations (Part I)*, Lisp and Symbolic Computation, Vol. 6, Nos. 3/4, pages 259–288, 1993. Earlier version available in the proceedings of an ACM Conference on Proving Assertions about Programs, SIGPLAN Notices, Vol. 7, No. 1, January 1972. Cited on page 38.

[86] Cormac Flanagan. Hybrid type checking. In Simon Peyton Jones, editor, *POPL '06: Proceedings of the 33rd annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 245–256, Charleston, South Carolina, January 2006. Cited on pages 96 and 156.

[87] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In David W. Wall, editor, *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Languages Design and Implementation*, pages 237–247, Albuquerque, New Mexico, June 1993. Cited on pages 35, 39, and 110.

[88] Matthew Flatt, Gang Yu, Robert Bruce Findler, and Matthias Felleisen. Adding delimited and composable control to a production programming environment. In Norman Ramsey, editor, *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP'07)*, pages 165–176, Freiburg, Germany, October 2007. Cited on page 38.

[89] Matthew Fluet and Stephen Weeks. Contification using dominators. In Xavier Leroy, editor, *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP'01)*, pages 2–13, Firenze, Italy, September 2001. Cited on pages 35 and 44.

[90] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995. Cited on page 148.

[91] Steven E. Ganz, Daniel P. Friedman, and Mitchell Wand. Trampolined style. In Didier Rémi and Peter Lee, editors, *Proceedings of the Forth ACM SIGPLAN International Conference on Functional Programming (ICFP'99)*, pages 18–27, Paris, France, September 1999. Cited on page 42.

[92] Kathrin Geilmann and Arnd Poetzsch-Heffter. Modular Checking of Confinement for Object-Oriented Components using Abstract Interpretation. In *IWACO '11: International Workshop on Aliasing, Confinement and Ownership in object-oriented programming*, 2011. Cited on pages 154 and 161.

[93] Jean-Yves Girard. Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination de coupures dans l'analyse et la théorie des types. In Jens Erik Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, pages 63–92. North-Holland Publishing Co., 1971. Cited on pages 14 and 31.

[94] Donald Gordon and James Noble. Dynamic ownership in a dynamic language. In Pascal Costanza and Robert Hirschfeld, editors, *DLS '07: Proceedings of the 2007 symposium on Dynamic languages*, pages 41–52, Montreal, Quebec, Canada, 2007. ACM. Cited on pages 154 and 157.

[95] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java Language Specification, The 3rd Edition*. Addison-Wesley Professional, 2005. Cited on pages 33 and 98.

[96] Michael Greenberg, Benjamin C. Pierce, and Stephanie Weirich. Contracts made manifest. In Jens Palsberg, editor, *POPL '10: Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 353–364, Madrid, Spain, 2010. Cited on page 157.

[97] David Greenfieldboyce and Jeffrey S. Foster. Type qualifier inference for Java. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems, languages and applications*, pages 321–336, Montreal, Quebec, Canada, 2007. ACM. Cited on page 158.

[98] Chris Hankin and Daniel Le Métayer. Deriving Algorithms from Type Inference Systems: Application to Strictness Analysis. In Hans-J. Boehm, editor, *POPL '94: Proceedings of the 21st Annual ACM Symposium on Principles of programming languages*, pages 202–212, Portland, Oregon, January 1994. Cited on pages xx, 5, 11, 13, 14, 71, 72, 73, and 87.

[99] John Hannan and Dale Miller. From operational semantics to abstract machines. In Mitchell Wand, editor, *Special issue on the 1990 ACM Conference on Lisp and Functional Programming*, Mathematical Structures in Computer Science, Vol. 2, No. 4, pages 415–459. Cambridge University Press, December 1992. Cited on page 72.

[100] John Hatcliff and Olivier Danvy. A generic account of continuation-passing styles. In Hans-J. Boehm, editor, *POPL '94: Proceedings of the 21st Annual ACM Symposium on Principles of programming languages*, pages 458–471, Portland, Oregon, January 1994. Cited on page 35.

[101] Rich Hickey. The Clojure programming language. In Johan Brichau, editor, *Proceedings of the 2008 Symposium on Dynamic Languages (DLS '08)*, page 1, Paphos, Cyprus, 2008. ACM. Cited on page 42.

[102] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969. Cited on page 20.

[103] Wei Huang, Werner Dietl, Ana Milanova, and Michael D. Ernst. Inference and checking of object ownership. In James Noble, editor, *ECOOP 2012: Proceedings of the 26th European conference on Object-Oriented Programming*, Lecture Notes in Computer Science, Bejing, China, 2012. Springer-Verlag. Cited on page 161.

[104] Wei Huang and Ana Milanova. Towards efective inference and checking of ownership types. In *IWACO '11: International Workshop on Aliasing, Confinement and Ownership in object-oriented programming*, Lancaster, UK, 2011. Cited on pages xxi, 160, and 161.

[105] Gérard P. Huet. The Zipper. *Journal of Functional Programming*, 7(5):549–554, 1997. Cited on pages 41 and 58.

[106] John Hughes. Why functional programming matters. *The Computer Journal - Special issue on Lazy functional programming*, 32(2):98–107, April 1989. Cited on page 31.

[107] Graham Hutton. Higher-order functions for parsing. *Journal of Functional Programming*, 2(3):323–343, 1992. Cited on page 32.

[108] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, May 2001. Cited on page 104.

[109] Lintaro Ina and Atsushi Igarashi. Gradual typing for generics. In *OOPSLA '11: Proceedings of the 26th annual ACM SIGPLAN conference on Object-oriented programming systems, languages and applications*, pages 609–624, Portland, Oregon, USA, 2011. ACM. Cited on pages 96 and 156.

[110] JetBrains Inc. IntelliJ IDEA 11.1 Web Help. Refactoring Source Code. Online, 2012. Cited on page 43.

[111] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *Proceedings of NFM, NASA Formal Methods - Third International Symposium*, volume 6617 of *Lecture Notes in Computer Science*, pages 41–55, Pasadena, CA, USA, 2011. Springer. Cited on page 21.

[112] Suresh Jagannathan, Peter Thiemann, Stephen Weeks, and Andrew Wright. Single and loving it: must-alias analysis for higher-order languages. In Luca Cardelli, editor, *POPL '98: Proceedings of the 25th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 329–341, San Diego, California, January 1998. Cited on page 159.

[113] Thomas P. Jensen. Strictness Analysis in Logical Form. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*, pages 352–366, London, UK, 1991. Springer-Verlag. Cited on pages 14 and 87.

[114] Thomas Johnsson. Lambda lifting: Transforming programs to recursive equations. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 190–203, Nancy, France, September 1985. Springer-Verlag. Cited on page 43.

[115] Richard A. Kelsey. A correspondence between continuation passing style and static single assignment form. *ACM SIGPLAN Notices*, 30(3):13–22, March 1995. Cited on page 35.

[116] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An Overview of AspectJ. In Jørgen Lindskov Knudsen, editor, *Proceedings of the 15th European Conference on Object-Oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353, Budapest, Hungary, 2001. Springer. Cited on page 91.

[117] Oleg Kiselyov, Chung chieh Shan, and Amr Sabry. Delimited dynamic binding. In Julia Lawall, editor, *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming (ICFP'06)*, pages 26–37, Portland, Oregon, September 2006. Cited on page 24.

[118] Donald E. Knuth. The genesis of attribute grammars. volume 461 of *Lecture Notes in Computer Science*, pages 1–12, Paris, France, 1990. Springer. Cited on page 90.

[119] Neelakantan R. Krishnaswami and Jonathan Aldrich. Permission-based ownership: encapsulating state in higher-order typed languages. In Mary Hall, editor, *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 96–106, Chicago, IL, USA, 2005. Cited on page 164.

[120] Jean-Louis Krivine. A call-by-name lambda-calculus machine. *Higher-Order and Symbolic Computation*, 20(3):199–207, 2007. Cited on page 92.

[121] George Kuan. A rewriting semantics for type inference. Master's thesis, University of Chicago, 2007. Cited on pages 11 and 14.

[122] George Kuan. Type Checking and Inference via Reductions. [79], pages 403–428. Cited on pages 14, 52, and 90.

[123] George Kuan, David MacQueen, and Robert Bruce Findler. A Rewriting Semantics for Type Inference. In Rocco De Nicola, editor, *ESOP '07: Proceedings of the 16th European Symposium on Programming Languages and Systems*, volume 4421, pages 426–440, Braga, Portugal, 2007. Springer-Verlag. Cited on pages 7, 13, 14, 52, 75, 86, and 91.

[124] Peter J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964. Cited on pages 9, 14, 26, 31, 32, 36, 71, and 72.

[125] Peter J. Landin. A correspondence between Algol 60 and Church's lambda notation. *Communications of the ACM*, 8:89–101 and 158–165, 1965. Cited on pages 31 and 36.

[126] Peter J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, 1966. Cited on page 11.

[127] Peter J. Landin. A Generalization of Jumps and Labels. *Higher-Order and Symbolic Computation*, 11(2):125–143, 1998. Cited on page 36.

[128] Xavier Leroy. Coinductive big-step operational semantics. In Peter Sestoft, editor, *ESOP 2006: Programming Languages and Systems, 15th European Symposium on Programming*, volume 3924 of *Lecture Notes in Computer Science*, pages 54–68, Vienna, Austria, 2006. Springer. Cited on page 23.

[129] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The Objective Caml system, documentation and user's manual – release 3.12*. INRIA, July 2011. Cited on page 31.

[130] Yi Lu and John Potter. On ownership and accessibility. In Dave Thomas, editor, *ECOOP 2006: Proceedings of the 20th European conference on Object-Oriented Programming*, volume 4067 of *Lecture Notes in Computer Science*, pages 99–123, Nantes, France, 2006. Springer-Verlag. Cited on pages 148 and 157.

[131] Kin-Keung Ma and Jeffrey S. Foster. Inferring aliasing and encapsulation properties for Java. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems, languages and applications*, pages 423–440, Montreal, Quebec, Canada, 2007. ACM. Cited on pages 155 and 161.

[132] Conor McBride and James McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004. Cited on pages 8 and 92.

[133] Erik Meijer, Maarten M. Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In John Hughes, editor, *Proceedings of the Fifth ACM Conference on Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 124–144, Cambridge, Massachusetts, August 1991. Springer-Verlag. Cited on page 41.

[134] José Meseguer and Grigore Roşu. The rewriting logic semantics project. *Theoretical Computer Science*, 373(3):213–237, 2007. Cited on page 91.

[135] Bertrand Meyer. Design by Contract: The Eiffel Method. In *TOOLS (26)*, page 446, Santa Barbara, CA, USA, 1998. IEEE Computer Society. Cited on page 164.

[136] Jan Midtgaard. *Transformation, Analysis, and Interpretation of Higher-Order Procedural Programs*. PhD thesis, BRICS PhD School, University of Aarhus, Aarhus, Denmark, June 2007. Cited on pages 13 and 85.

[137] Jan Midtgaard and Thomas Jensen. A calculational approach to control-flow analysis by abstract interpretation. In María Alpuente and Germán Vidal, editors, *Static Analysis, 15th International Symposium, SAS 2008*, volume 5079 of *Lecture Notes in Computer Science*, pages 347–362, Valencia, Spain, July 2008. Springer-Verlag. Cited on pages 29 and 34.

[138] Matthew Might. *Environment analysis of higher-order languages*. PhD thesis, Georgia Institute of Technology, Atlanta, GA, USA, 2007. Cited on pages 34 and 40.

[139] Matthew Might. Abstract interpreters for free. In Radhia Cousot and Matthieu Martel, editors, *SAS'10: Proceedings of the 17th international conference on Static analysis*, volume 6337 of *Lecture Notes in Computer Science*, pages 407–421, Perpignan, France, 2010. Springer-Verlag. Cited on pages 29, 33, 34, and 85.

[140] Matthew Might. Shape analysis in the absence of pointers and structure. In Gilles Barthe and Manuel V. Hermenegildo, editors, *VMCAI 2010: Proceedings of the 11th International Conference on Verification, Model Checking, and Abstract Interpretation*, volume 5944 of *Lecture Notes in Computer Science*, pages 263–278, Madrid, Spain, January 2010. Springer-Verlag. Cited on page 40.

[141] Matthew Might, Yannis Smaragdakis, and David Van Horn. Resolving and exploiting the k-CFA paradox: illuminating functional vs. object-oriented program analysis. In Alex Aiken, editor, *Proceedings of the ACM SIGPLAN 2010 Conference on Programming Languages Design and Implementation*, pages 305–315, Toronto, Canada, June 2010. Cited on page 29.

[142] Ana Milanova and Yin Liu. Practical static ownership inference. Technical report, Rensselaer Polytechnic Institute, Troy NY 12110, USA, 2010. Cited on page 159.

[143] Ana Milanova and Jan Vitek. Static dominance inference. In Judith Bishop and Antonio Vallecillo, editors, *Proceedings of the 49th international conference on Objects, models, components, patterns (TOOLS 2011)*, volume 6705 of *Lecture Notes in Computer Science*, pages 211–227, Zurich, Switzerland, 2011. Springer-Verlag. Cited on pages 96, 154, 160, and 161.

[144] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978. Cited on pages 5 and 14.

[145] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997. Cited on pages 15 and 31.

[146] Nick Mitchell. The runtime structure of object ownership. In Dave Thomas, editor, *ECOOP 2006: Proceedings of the 20th European conference on Object-Oriented Programming*, volume 4067 of *Lecture Notes in Computer Science*, pages 74–98, Nantes, France, 2006. Springer-Verlag. Cited on page 158.

[147] Samuel E. Moelius III and Amie L. Souter. An object ownership inference algorithm and its applications. In *MASPLAS '04: Mid-Atlantic Student Workshop on Programming Languages and Systems*, 2004. Cited on pages 96, 159, and 161.

[148] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991. Cited on page 35.

[149] Steven S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997. Cited on page 35.

[150] Peter Müller. VSTTE 2005: Verified Software: Theories, Tools, Experiments: Zurich, Switzerland, October 10-13, 2005, Revised Selected Papers and Discussions. chapter Reasoning about Object Structures Using Ownership, pages 93–104. Springer-Verlag, Berlin, Heidelberg, 2008. Cited on page 95.

[151] Andrew C. Myers. JFlow: Practical Mostly-Static Information Flow Control. In Alex Aiken, editor, *POPL '99: Proceedings of the 26th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 228–241, San Antonio, Texas, January 1999. Cited on page 97.

[152] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999. Cited on page 5.

[153] Hanne Riis Nielson and Flemming Nielson. *Semantics with applications: a formal introduction*. John Wiley & Sons, Inc., New York, NY, USA, 1992. Cited on pages 20, 21, and 34.

[154] Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007. Cited on pages 8 and 92.

[155] Nathaniel Nystrom, Stephen Chong, and Andrew C. Myers. Scalable extensibility via nested inheritance. In *OOPSLA '04: Proceedings of the 19th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 99–115, Vancouver, British Columbia, Canada, 2004. Cited on page 135.

[156] Martin Odersky. *The Scala Language Specification, version 2.9*. École Polytechnique Fédérale de Lausanne, 2012. Cited on pages 31 and 87.

[157] Martin Odersky, Vincent Cremet, Christine Röckl, and Matthias Zenger. A Nominal Theory of Objects with Dependent Types. In Luca Cardelli, editor, *ECOOP 2003: Proceedings of the 18th European conference on Object-Oriented Programming*, volume 2743 of *Lecture Notes in Computer Science*, pages 201–224, Darmstadt, Germany, 2003. Springer-Verlag. Cited on page 103.

[158] Atsushi Ohori and Isao Sasano. Lightweight Fusion by Fixed Point Promotion. In Matthias Felleisen, editor, *POPL '07: Proceedings of the 43rd annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 143–154, Nice, France, January 2007. Cited on pages 14 and 42.

[159] Johan Östlund and Tobias Wrigstad. Welterweight Java. In Jan Vitek, editor, *Proceedings of the 48th international conference on Objects, models, components, patterns (TOOLS 2010)*, volume 6141 of *Lecture Notes in Computer Science*, pages 97–116, Berlin, Heidelberg, 2010. Springer-Verlag. Cited on pages 114 and 127.

[160] Johan Östlund and Tobias Wrigstad. Owners as Ombudsmen. In *IWACO '11: International Workshop on Aliasing, Confinement and Ownership in object-oriented programming*, Lancaster, UK, 2011. Cited on page 153.

[161] Nicolas Oury and Wouter Swierstra. The power of Pi. In Peter Thiemann, editor, *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP'08)*, pages 39–50, Victoria, British Columbia, Canada, September 2008. Cited on page 8.

[162] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987. Cited on pages 17, 32, and 44.

[163] Simon L. Peyton Jones, editor. *Haskell 98 Language and Libraries: the Revised Report*. Cambridge University Press, 2003. Cited on pages 24 and 31.

[164] Benjamin C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002. Cited on pages xix, 14, 20, 21, 24, 33, 70, 75, 90, 115, and 116.

[165] Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975. Cited on page 38.

[166] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report FN-19, Department of Computer Science, University of Aarhus, Aarhus, Denmark, September 1981. Cited on pages 20 and 23.

[167] Gordon D. Plotkin. The origins structural operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:3–15, July-December 2004. Cited on page 20.

[168] Arnd Poetzsch-Heffter, Kathrin Geilmann, and Jan Schäfer. Infering Ownership Types for Encapsulated Object-Oriented Program Components. In Thomas W. Reps, Mooly Sagiv, and Jörg Bauer, editors, *Program Analysis and Compilation*, volume 4444 of *Lecture Notes in Computer Science*, pages 120–144. Springer, 2007. Cited on page 161.

[169] Alex Potanin, James Noble, and Robert Biddle. Checking ownership and confinement. *Concurrency and Computation: Practice and Experience*, 16(7):671–687, June 2004. Cited on page 158.

[170] Alex Potanin, James Noble, Dave Clarke, and Robert Biddle. Generic ownership for generic Java. In *OOPSLA '06: Proceedings of the 21st ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 311–324, Portland, Oregon, USA, 2006. Cited on pages 138, 144, and 147.

[171] Xin Qi and Andrew C. Myers. Masked types for sound object initialization. In Benjamin C. Pierce, editor, *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 53–65, Savannah, GA, USA, 2009. Cited on page 96.

[172] Vincent Rahli. *Investigations in intersection types: Confluence, and semantics of expansion in the λ-calculus, and a type error slicing method*. PhD thesis, Heriot-Watt University, Edinburgh, Scotland, UK, March 2011. Cited on page 87.

[173] Chris Rathman. Clojure: Trampoline for mutual recursion. Lambda the Ultimate, available at `http://lambda-the-ultimate.org/node/3106`. 26th November 2008. Cited on page 42.

[174] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of 25th ACM National Conference*, pages 717–740, Boston, Massachusetts, 1972. Reprinted in Higher-Order and Symbolic Computation 11(4):363–397, 1998. Cited on pages 11, 13, 40, 43, 72, 77, and 89.

[175] John C. Reynolds. Towards a theory of type structure. In B. Robinet, editor, *Programming Symposium*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425, Berlin, 1974. Springer-Verlag. Cited on pages 14 and 31.

[176] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of 17th IEEE Symposium on Logic in Computer Science (LICS 2002)*, pages 55–74, Copenhagen, Denmark,, 2002. IEEE Computer Society. Cited on page 21.

[177] Tiark Rompf, Ingo Maier, and Martin Odersky. Implementing first-class polymorphic delimited continuations by a type-directed selective CPS-transform. In Graham Hutton and Andrew P. Tolmach, editors, *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP'09)*, pages 317–328, Edinburgh, Scotland, Aug–Sep 2009. Cited on page 38.

[178] Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. In Carolyn L. Talcott, editor, *Special issue on continuations (Part I)*, Lisp and Symbolic Computation, Vol. 6, Nos. 3/4, pages 289–360, 1993. Cited on page 35.

[179] Amr Sabry and Matthias Felleisen. Is continuation-passing useful for data flow analysis? In Vivek Sarkar, editor, *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Languages Design and Implementation*, pages 1–12, Orlando, Florida, June 1994. Cited on page 34.

[180] Dana Scott and Christopher Strachey. Toward A Mathematical Semantics for Computer Languages. In Jerome Fox, editor, *Proceedings of the Symposium on Computers and Automata*, pages 19–46, Brooklyn, NY, USA, April 1971. Polytechnic Press. Cited on page 20.

[181] Ilya Sergey and Dave Clarke. From type checking by recursive descent to type checking with an abstract machine. In Claus Brabrand and Eric Van Wyk, editors, *LDTA '11: Proceedings of the 11th International Workshop on Language Descriptions, Tools, and Applications*, pages 9–15, Saarbrücken, Germany, 2011. Cited on pages 17 and 56.

[182] Ilya Sergey and Dave Clarke. A correspondence between type checking via reduction and type checking via evaluation. Accompanying code overview. Technical Report Report CW 617, Katholieke Universiteit Leuven, January 2012. Cited on page 17.

[183] Ilya Sergey and Dave Clarke. A correspondence between type checking via reduction and type checking via evaluation. *Information Processing Letters*, 112(1-2):13–20, January 2012. Cited on pages 17 and 75.

[184] Ilya Sergey, Dominique Devriese, Matthew Might, Jan Midtgaard, David Darais, Dave Clarke, and Frank Piessens. Monadic Abstract Interpreters. Unpublished draft, 2012. Cited on pages 29 and 34.

[185] Ilya Sergey, Jan Midtgaard, and Dave Clarke. Dominance analysis via ownership types and abstract interpretation. Technical Report Report CW 615, Katholieke Universiteit Leuven, December 2011. Cited on page 164.

[186] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages or Taming Lambda*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, May 1991. Technical Report CMU-CS-91-145. Cited on pages 34 and 40.

[187] Filip Sieczkowski, Małgorzata Biernacka, and Dariusz Biernacki. Automating derivations of abstract machines from reduction semantics: a generic formalization of refocusing in Coq. In *Proceedings of the 22nd international conference on Implementation and application of functional languages*, IFL'10, pages 72–88, Alphen aan den Rijn, The Netherlands, 2011. Springer-Verlag. Cited on page 92.

[188] Jeremy Siek and Walid Taha. Gradual typing for functional languages. In *Scheme '06: International Workshop on Scheme and Functional Programming*, pages 81–92, 2006. Cited on pages 96, 148, and 156.

[189] Jeremy Siek and Walid Taha. Gradual typing for objects. In Erik Ernst, editor, *ECOOP 2007: Proceedings of the 21st European conference on Object-Oriented Programming*, volume 4609 of *Lecture Notes in Computer Science*, pages 2–27, Berlin, Germany, 2007. Springer-Verlag. Cited on pages 96, 115, 120, 122, and 156.

[190] Jeffrey Mark Siskind. Flow-directed lightweight closure conversion. Technical Report 99-190R, NEC Research Institute, Inc., Dec 1999. Cited on page 43.

[191] Christian Skalka, Scott Smith, and David Van Horn. Types and trace effects of higher order programs. *Journal of Functional Programming*, 18(2):179–249, 2008. Cited on page 5.

[192] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. Pick your contexts well: understanding object-sensitivity. In Mooly Sagiv, editor, *POPL '11: Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 17–30, Austin, Texas, USA, 2011. Cited on page 29.

[193] Emma Söderberg and Görel Hedin. Building semantic editors using JastAdd: tool demonstration. In Claus Brabrand and Eric Van Wyk, editors, *LDTA '11: Proceedings of the 11th International Workshop on Language Descriptions, Tools, and Applications*, pages 81–86, Saarbrücken, Germany, 2011. Cited on page 165.

[194] Paul A. Steckler and Mitchell Wand. Lightweight closure conversion. *ACM Transactions on Programming Languages and Systems*, 19(1):48–86, 1997. Cited on page 43.

[195] Eijiro Sumii. An implementation of transparent migration on standard Scheme. In Matthias Felleisen, editor, *Proceedings of the Workshop on Scheme and Functional Programming*, pages 61–64, Montréal, Canada, September 2000. Rice Technical Report 00-368. Cited on page 38.

[196] Gerald J. Sussman and Guy L. Steele Jr. Scheme: An interpreter for extended lambda calculus. AI Memo 349, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, December 1975. Reprinted in Higher-Order and Symbolic Computation 11(4):405–439, 1998. Cited on pages 31 and 33.

[197] Wouter Swierstra. From Mathematics to Abstract Machine: A formal derivation of an executable Krivine machine. In *Proceedings of the fourth workshop on Mathematically Structured Functional Programming*, volume 76 of *EPTCS*, pages 163–177, Tallinn, Estonia, 2012. Open Publishing Association. Cited on page 92.

[198] Don Syme, Adam Granicz, and Antonio Cisternino. *Expert F# 2.0*. Apress, Berkley, CA, USA, 1st edition, 2010. Cited on page 31.

[199] Éric Tanter. Execution levels for aspect-oriented programming. In Mario Südholt, editor, *AOSD '10: Proceedings of the 9th International Conference on Aspect-Oriented Software Development*, pages 37–48, Rennes and Saint-Malo, France, 2010. ACM. Cited on page 91.

[200] Peter Thiemann. Combinators for program generation. *Journal of Functional Programming*, 9(5):483–525, 1999. Cited on page 38.

[201] Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3:121–189, 1995. Cited on page 87.

[202] Sam Tobin-Hochstadt and Matthias Felleisen. The Design and Implementation of Typed Scheme. In George C. Necula and Philip Wadler, editors, *POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 395–406, San Francisco, California, January 2008. Cited on pages 15 and 148.

[203] Sam Tobin-Hochstadt and David Van Horn. Higher-Order Symbolic Execution via Contracts. In *OOPSLA '12: Proceedings of the 27th annual ACM SIGPLAN conference on Object-oriented programming systems, languages and applications*, pages 537–554, Tucson, Arizona, USA, 2012. ACM. Cited on page 164.

[204] David Van Horn and Matthew Might. Abstracting Abstract Machines. In Stephanie Weirich, editor, *ICFP '10: Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, pages 51–62,

Baltimore, Maryland, USA, September 2010. ACM. Cited on pages 29, 33, and 85.

[205] Jan Vitek and Boris Bokowski. Confined types. In A. Michael Berman, editor, *OOPSLA '99: Proceedings of the 14th annual ACM SIGPLAN conference on Object-oriented programming systems, languages and applications*, pages 82–96, Denver, Colorado, USA, 1999. ACM. Cited on page 95.

[206] Philip Wadler. Deforestation: Transforming Programs to Eliminate Trees. *Theoretical Computer Science*, 73(2):231–248, 1990. Cited on page 41.

[207] Philip Wadler and Robert Bruce Findler. Well-Typed Programs Can't Be Blamed. In Giuseppe Castagna, editor, *ESOP '09: Proceedings of the 18th European Symposium on Programming Languages and Systems*, volume 5502 of *Lecture Notes in Computer Science*, pages 1–16, York, UK, 2009. Springer-Verlag. Cited on pages 155, 156, and 164.

[208] Stephen Weeks. Whole-program compilation in MLton. In Andrew Kennedy and François Pottier, editors, *ML '06:Proceedings of the ACM SIGPLAN 2006 workshop on ML*, page 1, September 2006. Invited talk. Cited on page 44.

[209] Glynn Winskel. *The Formal Semantics of Programming Languages*. Foundation of Computing Series. The MIT Press, 1993. Cited on page 20.

[210] Roger Wolff, Ronald Garcia, Éric Tanter, and Jonathan Aldrich. Gradual typestate. In Mira Mezini, editor, *ECOOP 2011: Proceedings of the 25th European conference on Object-Oriented Programming*, volume 6831 of *Lecture Notes in Computer Science*, pages 459–483, Lancaster, UK, 2011. Springer-Verlag. Cited on page 156.

[211] Alisdair Wren. Inferring ownership. Master's thesis, Imperial College London, UK, June 2003. Cited on page 158.

[212] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115:38–94, November 1994. Cited on pages 5, 97, and 130.

[213] Tobias Wrigstad and Dave Clarke. Existential owners for ownership types. *Journal of Object Technology*, 6(4), 2007. Cited on page 157.

[214] Tobias Wrigstad and Dave Clarke. Is the World Ready for Ownership Types? Is Ownership Types Ready for the World? In *IWACO '11: International Workshop on Aliasing, Confinement and Ownership in object-oriented programming*, Lancaster, UK, 2011. Cited on page 148.

[215] Tobias Wrigstad, Francesco Zappa Nardelli, Sylvain Lebresne, Johan Östlund, and Jan Vitek. Integrating typed and untyped code in a scripting language. In Jens Palsberg, editor, *POPL '10: Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 377–388, Madrid, Spain, 2010. Cited on page 156.

[216] Damiano Zanardini. Higher-order abstract non-interference. In Pawel Urzyczyn, editor, *Typed Lambda Calculi and Applications*, volume 3461 of *Lecture Notes in Computer Science*, pages 417–432. Springer Berlin / Heidelberg, 2005. Cited on pages 5 and 88.

Curriculum Vitæ

[...] oh, and this is Ilya Sergey. He follows me on Twitter.
A big CS professor

Ilya Sergey was born in Leningrad, USSR (nowadays, Saint Petersburg, Russia) on June the 12th, 1986. He graduated from Saint Petersburg Lyceum 239 in 2003 and received the Master degree, summa cum laude, in Mathematics and Computer Science from Saint Petersburg State University in 2008. Before joining KU Leuven as a PhD student, Ilya worked as a software developer in JetBrains Inc.

Since November 2008 Ilya is a member of the DistriNet research group under supervision of Prof. Dave Clarke. His research interests dwell in the area of the design and implementation of programming languages. He is particularly interested in developing expressive type systems and static program analyses.

Ilya visited École Polytechnique Fédérale de Lausanne (hosted by Prof. Martin Odersky) for two weeks in 2009, where he worked on a type system for virtual classes in Scala. Ilya was a visiting PhD fellow at Aarhus University (hosted by Prof. Olivier Danvy) in 2010 and 2011 for three months in total, where he worked on numerous topics concerning inter-deriving program semantics and constructing static program analyses, in collaboration with Jan Midtgaard, Prof. Matthew Might, and David Van Horn. Ilya has been offered an intern position in Microsoft Research Cambridge for twelve weeks in summer 2012 under supervision of Simon Peyton Jones, where he worked on improving the demand analyser of Glasgow Haskell Compiler.

List of Publications

Journal Articles

- Ilya Sergey and Dave Clarke. A correspondence between type checking via reduction and type checking via evaluation. *Information Processing Letters*, volume 112, issue 1-2, pages 13–20, January 2012. Elsevier.

International Conference Articles

- Christopher Earl, Ilya Sergey, Matthew Might and David Van Horn. Introspective Pushdown Analysis of Higher-Order Programs. In Robby Findler, editor, proceedings of the *17th ACM SIGPLAN International Conference on Functional Programming (ICFP 2012)*, pages 177–188. 10–12 September 2012. Copenhagen, Denmark. ACM.
- Ilya Sergey, Jan Midtgaard and Dave Clarke. Calculating Graph Algorithms for Dominance and Shortest Path. In Jeremy Gibbons and Pablo Nogueira, editors, proceedings of the *11th International Conference on Mathematics of Program Construction (MPC 2012)*, volume 7342 of *Lecture Notes in Computer Science*, pages 132–156. 25–27 June 2012. Madrid, Spain. Springer.
- Ilya Sergey and Dave Clarke. Gradual Ownership Types. In Helmut Seidl, editor, proceedings of the *21th European Symposium on Programming (ESOP 2012)*, volume 7211 of *Lecture Notes in Computer Science*, pages 579–599. 24 March – 1 April 2012. Tallinn, Estonia. Springer.

Peer-Reviewed International Workshop Articles

- Ilya Sergey and Dave Clarke. Towards Gradual Ownership Types. In *International Workshop on Aliasing, Confinement and Ownership (IWACO 2011)*. 25 July 2011. Lancaster, UK.
- Ilya Sergey and Dave Clarke. From type checking by recursive descent to type checking with an abstract machine. In Claus Brabrand and Eric Van Wyk, editors, proceedings of the *Eleventh Workshop on Language Descriptions, Tools and Applications (LDTA 2011)*, pages 1–7. 26–27 March 2011. Saarbrücken, Germany. ACM.
- Ilya Sergey, Dave Clarke and Alexander Podkhalyuzin. Automatic refactorings for Scala programs. *Scala Days 2010 Workshop*. April 2010. EPFL, Lausanne, Switzerland. Available as a Technical Report in *CW Reports, volume CW577*, Department of Computer Science, KU Leuven.
- Dave Clarke and Ilya Sergey. A semantics for context-oriented programming with layers. In *International Workshop on Context-Oriented Programming (COP 2009)*, pages 1–6. June 2009. Genova, Italy. ACM.

Book Chapters

- Dave Clarke, Johan Östlund, Ilya Sergey and Tobias Wrigstad. Ownership Types: A Survey. In Dave Clarke, James Noble and Tobias Wrigstad, editors, *Aliasing in Object-Oriented Programming*. To appear.

Technical Reports

- Ilya Sergey and Dave Clarke. A correspondence between type checking via reduction and type checking via evaluation. Accompanying code overview. *CW Reports, volume CW617*, 20 pages, Department of Computer Science, KU Leuven. January 2012. Leuven, Belgium.
- Ilya Sergey, Jan Midtgaard and Dave Clarke. Dominance Analysis via Ownership Types and Abstract Interpretation. *CW Reports, volume CW615* 23 pages, Department of Computer Science, KU Leuven. December 2011. Leuven, Belgium.

- Ilya Sergey and Dave Clarke. Gradual Ownership Types. *CW Reports, volume CW613*, 43 pages, Department of Computer Science, KU Leuven. December 2011. Leuven, Belgium.
- Ilya Sergey, Peter Gromov and Dave Clarke. Scripting an IDE for EDSL awareness. *CW Reports, volume CW608*, 9 pages, Department of Computer Science, KU Leuven. July 2011. Leuven, Belgium.
- Ilya Sergey, Dave Clarke and Alexander Podkhalyuzin. Automatic refactorings for Scala programs. *CW Reports, volume CW577*, 6 pages, Department of Computer Science, KU Leuven. April 2010. Leuven, Belgium.

Other Manuscripts

- Ilya Sergey. Implementation of Gradual Ownership Types for Java using Attribute Grammars (in Russian, Реализация гибридных типов владения в Java посредством атрибутных грамматик). In Andrey Terekhov, Dmitry Boulytchev, editors, *Software Engineering*, issue 6, pages 49–79, Saint Petersburg State University publishing, 2011.
- Ilya Sergey and Andrey Barabanov. Extraction of musical notation from musical signal (in Russian, Получение нотной записи одноголосного музыкального сигнала). In *Phonetical Lyceum*, issue 4, Faculty of Philology and Arts, Saint Petersburg State University publishing, 2009.
- Ilya Sergey. Implementation of JVM-based languages support in IntelliJ IDEA. In *International Workshop Workshop on Multiparadigm Programming with Object-Oriented Languages (MPOOL 2008)*. July 2008. Paphos, Cyprus.

Arenberg Doctoral School of Science, Engineering & Technology
Faculty of Engineering
Department of Computer Science
Scientific Computing Group
Celestijnenlaan 200A, box 2402
B-3001 Heverlee