THESIS ON INFORMATICS AND SYSTEM ENGINEERING C43

Logics for Low-Level Code and Proof-Preserving Program Transformations

Ando Saabas

Tallinn 2008

TALLINN UNIVERSITY OF TECHNOLOGY
Institute of Cybernetics

This dissertation was accepted for the defence of the degree of Doctor of Philosophy in Engineering on October 20, 2008.

Supervisors: Dr. Tarmo Uustalu
Institute of Cybernetics at Tallinn University of Technology

Enn Tõugu, DSc
Institute of Cybernetics at Tallinn University of Technology

Opponents: Dr. Bernd Fischer
School of Electronics and Computer Science
University of Southampton

Prof. David Sands
Department of Computer Science and Engineering
Chalmers University of Technology

Defence: November 14, 2008

Declaration: Hereby I declare that this doctoral thesis, my original investigation and achievement, submitted for the doctoral degree at Tallinn University of Technology has not previously been submitted for any degree or examination.

/Ando Saabas/

Copyright: Ando Saabas, 2008
ISSN 1406-4731
ISBN 978-9985-59-865-8

INFORMAATIKA JA SÜSTEEMITEHNIKA C43

Loogikad madala taseme koodile ja tõestusi säilitavad programmiteisendused

Ando Saabas

Tallinn 2008

Logics for Low-Level Code and
Proof-Preserving Program Transformations

Abstract

The Proof-Carrying Code (PCC) paradigm has emerged as a way of instilling trust in the code user about the properties of the code that she is about to run. The underlying idea is simple: code is shipped with a proof which attests that it adheres to the requirements set at the user's computer. Consequently, the user does not need to check the code itself, only its proof, which is a simple, fast and one-time procedure. A program proof can be thought of as a semantic checksum, attesting that the semantics of the program has not been tampered with.

While the underlying idea of Proof-Carrying Code is simple, it offers many challenges in both scientific and engineering aspects. This thesis concentrates on two aspects relevant for Proof-Carrying Code. First of all, we describe a way of giving a compositional semantics and matching Hoare logic to low-level, "unstructured" languages with jumps. Our work is based on the insight that a phrase structure can be given to the seemingly non-modular code by defining the code to be either a single instruction, or a finite union of pieces of code. We show that this seemingly trivial phrase structure actually provides a convenient basis for compositional semantics and logic. The semantic and logic descriptions that we thus obtain are similar in sophistication to those of the standard WHILE language. Notably, Hoare triples in our logic can be interpreted in the usual way.

The second aspect we investigate concerns "proof compilation": the problem of translating a program proof alongside the program in the context of compilation. While the problem is trivial in the case of a non-optimizing compiler, it becomes complicated when optimizations take place: a valid proof of a program is in general not valid for the optimized, semantically equivalent version of the same program. We propose a way of describing optimizations via type systems, where the type system specifies both the dataflow analysis underlying the optimization and the rewrite rules making use of the analysis information and carrying out the optimization. The type derivation of a program is then used to guide the transformation of the proof. We demonstrate that this approach works both for high-level programs and Hoare proofs and on control flow graph based program descriptions and flat, unstructured program proofs. We are able to address complicated, program structure changing optimizations such as partial redundancy elimination and also optimizations based on bidirectional analysis.

Loogikad madala taseme koodile ja tõestusi säilitavad programmiteisendused

Lühikokkuvõte

Nn. tõestusega kood (Proof-Carrying Code ehk PCC) on uuenduslik viis anda garantiisid tarkvara turvalisuse ning funktsionaalsuse kohta. Idee on lihtne: kompileeritud programmiga on kaasas tõestus — sertifikaat, mida kontrollides on võimalik kindlaks teha, kas programm esitatud nõuetele vastab. Seega ei pea kasutaja kontrollima programmi ennast, vaid ainult tõestust, mis on kiire ja ühekordne protseduur ning toimub programmi kasutaja enda arvutis. Programmi tõestusest võib mõelda kui semantilisest kontrollsummast, mis näitab, et programmi semantikat ei ole ebasobivalt muudetud.

Antud töö käsitleb kahte PCC-ga lähedalt seotud probleemi. Kuna tarkvara levitatakse kompileerituna (nn. madala taseme kood), on oluline madala taseme koodi üle arutlemine, mis toimub läbi programmiloogika. Kompositsiooniliste loogikate loomist madala taseme keelte jaoks on seni peetud keeruliseks, kuna neil puudub kõrgkeeltele omane ilmutatud kujul fraasistruktuur. Antud töös näitame, et madala taseme koodile võib anda struktuuri, defineerides koodi kas üksiku instruktsioonina või kooditükkide lõpliku ühendina. Ilmneb, et see näiliselt lihtne struktuur on piisav kompositsioonilise semantika ja loogika kirjeldamiseks. Defineeritud semantika ja loogika on oma keerukuselt lähedased standardsetele WHILE keele semantikale ja loogikale.

Teine probleem, millele antud töö pühendub on nn. "tõestuste kompileerimine". Kuna tõestused koostatakse üldjuhul kõrgkeelsete programmide kohta, on PCC arhitektuuri puhul mõistlik teisendada kõrgkeelsete programmide tõestused kompileerimise käigus automaatselt madala taseme koodi tõestusteks. See ülesanne on suhteliselt lihtne juhul, kui kompilaator ei teosta optimeerimisi. Vastasel juhul osutub transleerimine keeruliseks, kuna üldjuhul optimeeritud programmi jaoks algne tõestus enam ei sobi. Käesolevas töös esitame viisi, kuidas andmevooanalüüsidel põhinevaid optimeerimisi on võimalik esitada läbi tüübisüsteemide, kus tüübisüsteem sisaldab nii analüüsi- kui ka optimeerimiskomponenti. Nii on võimalik tüübituletuspuu põhjal teostada nii programmi- kui tõestusteisendusi. Oma töös näitame, et see lähenemine töötab nii kõrgkeelsete programmide ja tõestuste jaoks (sh. keeruliste optimeerimiste nagu osaline liiasuse kõrvaldamine) kui ka madala taseme koodi ja mitte-modulaarsete tõestuste puhul.

Acknowledgments

First and foremost, I would like to express my gratitude to my supervisor Tarmo Uustalu. I can honestly say a better supervisor is hard to come by and feel really lucky to have been able to work with him during my studies. I have learned immensely from our collaboration and feel that his rigor and scientific integrity are truly something to strive for. But even more importantly, he has not only been a supervisor, but also a friend.

I wish to thank Enn Tõugu for recruiting me to the Institute of Cybernetics and for having been a mentor for me during the years here. His positive attitude and open mind have always been an inspiration for me.

The Institute of Cybernetics has been a wonderful place to work at and has truly allowed me to concentrate on research, a privilege not all PhD students can enjoy in Estonia. I wish to thank the administrative staff, especially Jaan Penjam, for providing such a student-friendly work-environment and all my colleagues who have made my work here so pleasurable.

I am grateful to my opponents, Dr. Bernd Fischer and Prof. David Sands for the time and hard work they invested in reading my thesis, their insightful comments and constructive criticism.

I found the general topic for my thesis research during my internship in INRIA Sophia Antipolis. I am grateful to Gilles Barthe and Tamara Rezk for the fruitful discussions and collaboration that led me to this. I thank Margus Veanes for inviting me to Redmond for an internship in Microsoft Research, which was a wonderful experience.

There are several organizations that have supported my research financially through different projects. The research was supported by EU FP6 integrated project MOBIUS, the Estonian Science Foundation grants 5567 and 6940 and the Estonian Doctoral School in ICT (2005-2008). I would also like to thank the EITSA Tiger University Plus programme for their scholarship and the Estonian Association of Information Technology and Telecommunications (ITL) for the Ustus Agur stipend.

Last but not least, I wish to thank my family, especially my mother and father, from whom I got the drive to learn new things and gain a better understanding of the world. I am deeply grateful to them for all the support and encouragement they have given me ever since I was a kid.

Contents

1 Introduction **11**
- 1.1 Proof-Carrying Code 12
- 1.2 Contributions 14
- 1.3 References to previously published work 15
- 1.4 Organization of the thesis 16

2 Preliminaries **17**
- 2.1 The high-level language While 17
  - 2.1.1 Syntax 17
  - 2.1.2 Natural semantics 17
  - 2.1.3 Hoare logic 17
- 2.2 The low-level language Push 18
- 2.3 Program logics for Push 20

3 A compositional approach to low-level languages **25**
- 3.1 Structured version of Push and its natural semantics 25
- 3.2 Hoare logic 33
- 3.3 A dip into type systems 43
- 3.4 Compilation 46
  - 3.4.1 Example 53
- 3.5 Related work 56
- 3.6 Conclusion 57

4 Proof-preserving program transformations **59**
- 4.1 Introduction 59
- 4.2 Dead code elimination 62
  - 4.2.1 Type system for live variables analysis 62
  - 4.2.2 Type system for dead code elimination 65
- 4.3 Common subexpression elimination 75
  - 4.3.1 Type system for available expressions analysis 75
  - 4.3.2 Type system for conditional partial anticipability analysis 76
  - 4.3.3 Type system for common subexpression elimination 78
- 4.4 Partial redundancy elimination 90
  - 4.4.1 Simple PRE 90
  - 4.4.2 Full PRE 105

4.5 Related work 117
4.6 Conclusion 119

5 Bytecode transformations **121**
5.1 Background 121
5.2 Dead code elimination 122
5.2.1 Dead stores elimination 124
5.2.2 Load-pop pairs elimination 132
5.3 Store/load+ elimination 139
5.3.1 Duplicating loads elimination 139
5.3.2 Store-load pairs elimination 144
5.4 Related work 154
5.5 Conclusion 155

6 Conclusions and future work **157**

Bibliography **159**

Chapter 1

Introduction

The issue of determining whether a given program is correct, i.e. adheres to the given set of requirements, has been a topic of interest since the advent of computers. There are numerous approaches to show that a program is fully or partially correct with respect to some specification, ranging from manual testing and code review to automated/intelligent testing to type systems, static analysis, model checking and formal verification. These approaches are by no means mutually exclusive, although there certainly is some hierarchy of what kinds of properties can be shown using each individual approach. Not surprisingly, the more thorough one wants to be with checking, the more sophisticated the tools need to be, and the more laborious the job is for the person doing the checking. What can be checked and how also depends on how the requirements are specified: specifications can range from simple documentation expressed in a natural language, to semi-formal descriptions such as UML diagrams, to fully formal code annotations in some mathematical language such as first- or higher-order logic.

The holy grail of program correctness checking has been full formal verification, going back to seminal papers by Floyd [29] and Hoare [30] in the late Sixties. In formal verification, the requirements a program is intended to fulfill are specified in some mathematical language, typically first-order logic. The programming language must have a formal semantics, so an abstract mathematical model of the program can be built, and rigorously proven to adhere to the specification. The proof process is typically not fully automatic, but requires the guidance of the developer.

Formal verification can give a full guarantee that the program has the desired properties (assuming the verification tools themselves are correct). The specifications can be very complex and can address many different properties such as code's safety/security, functionality, bounds on resource consumption etc. This is opposed to for example testing, where full guarantees can usually not be given, or type systems, where the domain of what can be checked is limited.

The goal of formally verifying programs has proved to be very elusive and full verification is still far from being mainstream. Still there has been steady progress in approaching this goal. There are numerous tools available for program verification, such as the Extended Static Checker for Java ESC/Java2 [18], the KeY tool [12], Krakatoa for Java [42] and Caduceus for C [28], both built on the Why platform

[27], the Loop tool [66] and Jack [7], to name a few.

Verification is typically performed on the source code by the developer, while the user only obtains the compiled binary of the code. A problem orthogonal to verification is how to convince a user that it is safe to run the code she just obtained—giving access to full source code to the user is not always desired, and even if it is, the user usually does not have the means or the knowledge to compile the program. Additionally, in general the user cannot fully trust her compiler. A traditional method for giving some assurance to the user has been signing the code digitally. A digital signature however only speaks about the origin of the code, not about the code itself. The origin might not be fully trusted or even when it is, it does not give a full guarantee that the code is actually safe. Would it be possible to convince the user that the code is safe even when it is coming from an untrusted source?

1.1 Proof-Carrying Code

Proof-carrying code (PCC), proposed by Necula [45], is a technique for safely executing code coming from an untrusted source. The general underlying idea is quite simple: the untrusted code producer must supply the code with a proof, which shows that the code adheres to the policy set at the host's site. The host can then simply check the proof against the code and the site's policy, and only run the code if the proof is correct. The particular policy is chosen by the host and can address a wide array of properties, such as code safety, security or functionality. Proof checking is a one-time procedure and there is no need to use cryptography or trust an outside agent.

PCC has a lot of potential uses where the trusted computing base is dynamic, such as mobile phones, PDA's, smartcards etc. Example applications include extensible operating systems, downloadable applets, safety-critical embedded controllers etc.

The main highlights of the PCC architecture are the following.

- It is small and lightweight on the host's side. The host does not have to verify the code itself, only check a purported proof.
- PCC can operate on native code binaries. There is no inherent need for sandboxing PCC applications.
- It is general, so in addition to basic safety properties, other properties like functionality, non-interference etc. can be expressed using the same approach.
- It operates at load time, so there is no overhead in runtime checking.
- PCC programs are tamper-proof. If the code and/or the proof are modified, then either the proof becomes invalid with respect to the code and the policy and the program is rejected, or if the proof is still valid, it means that the

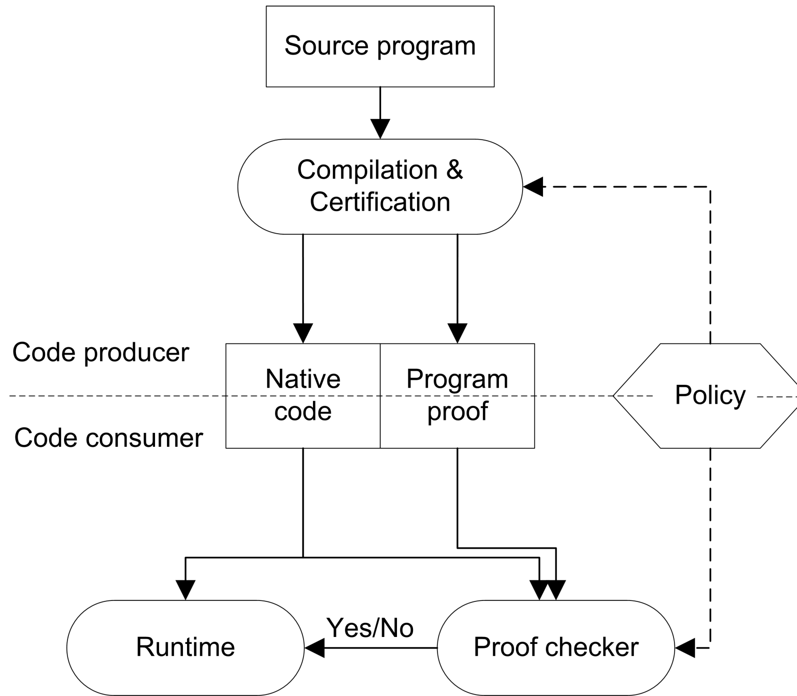


Figure 1.1: Simplified overview of the PCC architecture

program still adheres to the given policy despite of the modifications and is still safe to be executed.

Any instance of the general PCC architecture must contain at least a formal specification language used to express the safety/functionality policy, a formal semantics of the programming language of the distributed code, a formal language to express the proofs, and a proof checker, i.e., an algorithm validating the proofs. The general architecture of proof-carrying code is given in Figure 1.1.

While the basic idea underlying PCC is quite simple, there are many challenges in both theoretical and engineering aspects of PCC. Among those are:

- Finding suitable logics and type systems for reasoning about low-level code.
- Obtaining proofs for the low-level code from source-level proofs (especially in the light of compile-time optimizations).
- Compactly representing program proofs.
- Developing efficient and correct proof checkers.

All of this has made PCC a very popular topic in computer science, with a large amount of literature dedicated to it in the last ten years.

1.2 Contributions

This work concentrates on two related aspects relevant for PCC.

- The first aspect is related to finding suitable logics and type systems for reasoning about low-level code. There are are many challenges in this area; one question is how to deal with the unstructured nature of low-level code. Unlike high-level programs, low-level code has no explicit phrase structure due to the presence of jumps, i.e., it is seemingly non-modular. A consequence of a language being non-modular is that it cannot have compositional semantics, logics or type systems, therefore making reasoning about it more difficult. On the example of a bytecode-like language, we show that there is a way to give a useful structure to code with jumps and devise a compositional semantics and Hoare logic for low-level languages, which do not make any assumptions about the structure of the program. We show that the compositional semantics agrees to the standard semantics and prove the compositional Hoare logic to be sound and complete. The logic we present is standard in the sense that Hoare triples can be interpreted in the standard way. We also show that our approach allows for direct compilation from modular source-level program proofs into modular low-level proofs.

- The second aspect we deal with is related to the question of how to obtain certificates/proofs for compiled code when the source code is verified. Typically, programs are proved correct on the source level, using some verification environment. If the source code is compiled, one would like to "compile" its proof together with it. While this is straightforward for a non-optimizing compiler, this is not the case when optimizations take place: the proof has to be "optimized" along the program. We present a general and uniform way of transforming proofs, using type systems. The type system is a declarative representation of the dataflow analysis, and also includes an optimization component corresponding to the code rewrite rules based on the analysis result. The fundamental idea is to leverage the type system for optimizing both the program and its proof, i.e. a type derivation of a statement not only specifies how the statement has to be rewritten, but also how to modify its pre- and postconditions, so that they remain valid after the rewrite.

  We demonstrate our approach on three optimizations.

  - We introduce our technique on dead code elimination, a relatively simple optimization, requiring only one dataflow analysis. As such, it is a good example to describe our general approach. At the same time, it is interesting in being the archetypical optimization which requires weakening of assertions in program proofs, and is thus a counterexample to the popular belief that assertion transformation in the context of proof compilation is always strengthening.

- We look at common subexpression elimination, which is an optimization that requires linking of expression evaluation points to value reuse points and coordinated modifications of the program near both ends of such links, which seems to go against compositionality (where compositionality is what we want from a type system for a high-level language). We show how this can easily be overcome by a combined type system that reflects a combination of two analyses.
- Our work on high-level program transformation culminates on partial redundancy elimination, which is a very complex and subtle optimization that requires four interdependent dataflow analyses and performs code motion via edge splitting. In our type-systematic setting, this corresponds to introducing new code at subsumption inferences. On this example we also show that our approach is equally applicable to logics other than the standard Hoare logic, for example logics for reasoning about resource usage.

Besides showing the usefulness of the type-systematic approach for proof transformation, we argue that that this approach can be useful for other purposes. It can explain the optimization well in a declarative fashion, and makes it easy to show different properties of optimization like soundness and improvement (the latter is explained on the example of partial redundancy elimination).

We also extend our work on proof transformation to bytecode-like languages, where we use a flat, unstructured logic. Bytecode optimizations are interesting since many of them require bidirectional analyses (unlike most high-level optimizations). This means that during the analysis (and principal type inference), information needs to be propagated both back and forth in the control flow graph. This has the effect that the normal subsumption rule is not applicable, since changing the type at one program point potentially affects the type of all other program points. In our work, we show that our type-systematic approach scales equally well to unstructured logics and bidirectional dataflow analyses.

1.3 References to previously published work

Several parts of this thesis have been previously published as conference and journal papers.

Chapter 3 is based on papers "A Compositional Natural Semantics and Hoare Logic for Low-Level Languages" [53], published in *Theoretical Computer Science*, and "Compositional type systems for stack-based low-level languages" [52], presented at *Computing: Australasian Theory Symposium, CATS 2006*. It contains previously unpublished proofs of the metatheoretic properties of the compositional semantics and logic and the proof of preservation of Hoare triples.

Chapter 4 is based on papers “Program and proof optimizations with type systems” [55] published in the *Journal of Logic and Algebraic Programming* and “Proof optimization for partial redundancy elimination” [56], presented at *Partial Evaluation and Program Manipulation, PEPM 2008*.

Chapter 5 is based on a paper “Type Systems for Optimizing Stack-Based Code” [54], presented at *Bytecode Semantics, Verification, Analysis and Transformation, BYTECODE 2007*. The work on byte-code level proof transformations is previously unpublished.

All of these papers have roughly equal contributions from both co-authors.

1.4 Organization of the thesis

The rest of the thesis is organized as follows. In Chapter 2 we set the stage for our development by fixing the source and target languages, and the corresponding standard semantics and Hoare logic for these languages. In Chapter 3, we introduce a compositional semantics and Hoare logic for a low-level language with jumps. We also describe a compilation process to translate high-level proofs into corresponding low-level proofs. Chapter 4 explains the proof transformation method for program optimizations on the examples of dead code elimination, common subexpression elimination and partial redundancy elimination. Chapter 5 extends these ideas to optimizations on low-level languages and flat program proofs. Chapter 6 concludes and highlights some directions for future research.

CHAPTER 2

Preliminaries

In this chapter, we fix the syntax, natural semantics and the standard Hoare logic of the basic high-level language WHILE and its low-level counterpart, PUSH. The descriptions for WHILE are very well known and are given here for the sake of completeness; for a more thorough overview, the reader may turn to [48].

2.1 The high-level language While

2.1.1 Syntax

The syntax of WHILE proceeds from a countable supply of arithmetic variables $x \in \mathbf{Var}$. Over these, three syntactic categories of arithmetic expressions $a \in \mathbf{AExp}$, boolean expressions $b \in \mathbf{BExp}$ and statements $s \in \mathbf{Stm}$ are defined by means of the grammar

$$\begin{array}{rcl} a & ::= & x \mid n \mid a_0 + a_1 \mid \ldots \\ b & ::= & a_0 = a_1 \mid \ldots \mid \mathrm{tt} \mid \mathrm{ff} \mid \neg b \mid \ldots \\ s & ::= & x := a \mid \mathsf{skip} \mid s_0; s_1 \mid \mathsf{if}\ b\ \mathsf{then}\ s_0\ \mathsf{else}\ s_1 \mid \mathsf{while}\ b\ \mathsf{do}\ s \end{array}$$

2.1.2 Natural semantics

The semantics is given in terms of states. The states are defined as stores $\sigma \in \mathbf{Store}$, i.e., mappings of variables to integers: $\mathbf{State} =_{\mathrm{df}} \mathbf{Store} =_{\mathrm{df}} \mathbf{Var} \to \mathbb{Z}$. The arithmetical and boolean expressions are interpreted relative to stores as integers and truth values by the semantic function $[\![-]\!] \in \mathbf{AExp} + \mathbf{BExp} \to \mathbf{Store} \to \mathbb{Z}$, defined in the denotational style by the usual equations. We write $\sigma \models b$ to say that $[\![b]\!]\sigma = \mathrm{tt}$.

Statements are interpreted via the evaluation relation $\succ - \to \subseteq \mathbf{State} \times \mathbf{Stm} \times \mathbf{State}$ defined inductively by the ruleset given in Figure 2.1.

2.1.3 Hoare logic

The assertions $P \in \mathbf{Assn}$ are defined as formulae of an unspecified underlying logic over a signature consisting of (a) constants for integers and function and predicate

$$\frac{}{\sigma \succ x := a \to \sigma[x \mapsto [\![a]\!]\sigma]} :=_{\text{ns}} \qquad \frac{}{\sigma \succ \mathsf{skip} \to \sigma} \text{skip}_{\text{ns}} \qquad \frac{\sigma \succ s_0 \to \sigma'' \quad \sigma'' \succ s_1 \to \sigma'}{\sigma \succ s_0; s_1 \to \sigma'} \text{comp}_{\text{ns}}$$

$$\frac{\sigma \models b \quad \sigma \succ s_t \to \sigma'}{\sigma \succ \mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f \to \sigma'} \text{if}_{\text{ns}}^{\text{tt}} \qquad \frac{\sigma \not\models b \quad \sigma \succ s_f \to \sigma'}{\sigma \succ \mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f \to \sigma'} \text{if}_{\text{ns}}^{\text{ff}}$$

$$\frac{\sigma \models b \quad \sigma \succ s \to \sigma'' \quad \sigma'' \succ \mathsf{while}\ b\ \mathsf{do}\ s \to \sigma'}{\sigma \succ \mathsf{while}\ b\ \mathsf{do}\ s \to \sigma'} \text{while}_{\text{ns}}^{\text{tt}} \qquad \frac{\sigma \not\models b}{\sigma \succ \mathsf{while}\ b\ \mathsf{do}\ s \to \sigma} \text{while}_{\text{ns}}^{\text{ff}}$$

Figure 2.1: Natural semantics rules of WHILE

$$\frac{}{\{Q[a/x]\}\ x := a\ \{Q\}} :=_{\text{hoa}}$$

$$\frac{}{\{P\}\ \mathsf{skip}\ \{P\}} \text{skip}_{\text{hoa}} \qquad \frac{\{P\}\ s_0\ \{R\} \quad \{R\}\ s_1\ \{Q\}}{\{P\}\ s_0; s_1\ \{Q\}} \text{comp}_{\text{hoa}}$$

$$\frac{\{b \wedge P\}\ s_t\ \{Q\} \quad \{\neg b \wedge P\}\ s_f\ \{Q\}}{\{P\}\ \mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f\ \{Q\}} \text{if}_{\text{hoa}} \qquad \frac{\{b \wedge P\}\ s\ \{P\}}{\{P\}\ \mathsf{while}\ b\ \mathsf{do}\ s\ \{\neg b \wedge P\}} \text{while}_{\text{hoa}}$$

$$\frac{P \models P' \quad \{P'\}\ s\ \{Q'\} \quad Q' \models Q}{\{P\}\ s\ \{Q\}} \text{conseq}_{\text{hoa}}$$

Figure 2.2: Hoare rules of WHILE

symbols for the standard integer-arithmetical operations and relations and (b) the program variables $x \in \mathbf{Var}$ as constants. For the completeness result, the language is assumed to be expressive enough so as to allow the expression of the weakest liberal precondition of any statement wrt. any given postcondition (cf. [19]). We write $\sigma \models_\alpha P$ to express that P holds in the structure on $\mathbb{Z}$ determined by (a) the standard meanings of the arithmetical constants, function and predicate symbols and (b) a state σ, under an assignment α of the logical variables. The writing $P \models Q$ means that $\sigma \models_\alpha P$ implies $\sigma \models_\alpha Q$ for any σ, α.

The derivable judgements of the logic are given by the relation $\{\} - \{\} \subseteq \mathbf{Assn} \times \mathbf{Stm} \times \mathbf{Assn}$ defined inductively by the ruleset in Figure 2.2.

2.2 The low-level language Push

The low-level counterpart of WHILE we consider here is PUSH, a simple stack-based language with jumps. The building blocks of the syntax of PUSH are labels $\ell \in \mathbf{Label}$, which are natural numbers, and instructions $instr \in \mathbf{Instr}$. We also assume having a countable set of program variables (registers) $x \in \mathbf{Var}$ (just as in WHILE). The instructions of the language are defined by the grammar

$$\begin{array}{rcl} instr & ::= & \mathsf{load}\ x \mid \mathsf{store}\ x \mid \mathsf{push}\ n \mid \mathsf{add} \mid \mathsf{eq} \mid \ldots \\ & & \mid \mathsf{pop} \mid \mathsf{dup} \mid \mathsf{goto}\ \ell \mid \mathsf{gotoF}\ \ell \end{array}$$

$$\frac{(\ell, \mathsf{store}\ x) \in c \quad n \in \mathbb{Z}}{c \vdash (\ell, n :: zs, \sigma) \twoheadrightarrow (\ell+1, zs, \sigma[x \mapsto n])}\ \text{store} \qquad \frac{(\ell, \mathsf{load}\ x) \in c}{c \vdash (\ell, zs, \sigma) \twoheadrightarrow (\ell+1, \sigma(x) :: zs, \sigma)}\ \text{load}$$

$$\frac{(\ell, \mathsf{push}\ n) \in c}{c \vdash (\ell, zs, \sigma) \twoheadrightarrow (\ell+1, n :: zs, \sigma)}\ \text{push} \qquad \frac{(\ell, \mathsf{add}) \in c \quad n_0, n_1 \in \mathbb{Z}}{c \vdash (\ell, n_0 :: n_1 :: zs, \sigma) \twoheadrightarrow (\ell+1, n_0 + n_1 :: zs, \sigma)}\ \text{add}$$

$$\frac{(\ell, \mathsf{eq}) \in c \quad n_0, n_1 \in \mathbb{Z}}{c \vdash (\ell, n_0 :: n_1 :: zs, \sigma) \twoheadrightarrow (\ell+1, n_0 = n_1 :: zs, \sigma)}\ \text{eq}$$

$$\ldots$$

$$\frac{(\ell, \mathsf{pop}) \in c}{c \vdash (\ell, n :: zs, \sigma) \twoheadrightarrow (\ell+1, zs, \sigma)}\ \text{pop} \qquad \frac{(\ell, \mathsf{dup}) \in c}{c \vdash (\ell, n :: zs, \sigma) \twoheadrightarrow (\ell+1, n :: n :: zs, \sigma)}\ \text{dup}$$

$$\frac{(\ell, \mathsf{goto}\ m) \in c}{c \vdash (\ell, zs, \sigma) \twoheadrightarrow (m, zs, \sigma)}\ \text{goto}$$

$$\frac{(\ell, \mathsf{gotoF}\ m) \in c}{c \vdash (\ell, \mathrm{tt} :: zs, \sigma) \twoheadrightarrow (\ell+1, zs, \sigma)}\ \text{gotoF}^{\mathrm{tt}} \qquad \frac{(\ell, \mathsf{gotoF}\ m) \in c}{c \vdash (\ell, \mathrm{ff} :: zs, \sigma) \twoheadrightarrow (m, zs, \sigma)}\ \text{gotoF}^{\mathrm{ff}}$$

$$\frac{(\ell, \mathsf{store}\ x) \in c \quad \forall n \in \mathbb{Z}, zs' \in (\mathbb{Z}+\mathbb{B})^*.\ zs \neq n :: zs'}{c \vdash (\ell, zs, \sigma) \twoheadrightarrow\!\!|}\ \text{store}_{ab}$$

$$\frac{(\ell, \mathsf{add}) \in c \quad \forall n_0, n_1 \in \mathbb{Z}, zs' \in (\mathbb{Z}+\mathbb{B})^*.\ zs \neq n_0 :: n_1 :: zs'}{c \vdash (\ell, zs, \sigma) \twoheadrightarrow\!\!|}\ \text{add}_{ab}$$

$$\frac{(\ell, \mathsf{eq}) \in c \quad \forall n_0, n_1 \in \mathbb{Z}, zs' \in (\mathbb{Z}+\mathbb{B})^*.\ zs \neq n_0 :: n_1 :: zs'}{c \vdash (\ell, zs, \sigma) \twoheadrightarrow\!\!|}\ \text{eq}_{ab}$$

$$\frac{(\ell, \mathsf{pop}) \in c}{c \vdash (\ell, [], \sigma) \twoheadrightarrow\!\!|}\ \text{pop}_{ab} \qquad \frac{(\ell, \mathsf{dup}) \in c}{c \vdash (\ell, [], \sigma) \twoheadrightarrow\!\!|}\ \text{dup}_{ab}$$

$$\frac{\forall b \in \mathbb{B}, zs' \in (\mathbb{Z}+\mathbb{B})^*.\ zs \neq b :: zs'}{c \vdash (\ell, zs, \sigma) \twoheadrightarrow\!\!|}\ \text{gotoF}_{ab}$$

Figure 2.3: Single-step reduction rules of PUSH

(The pop and dup instructions are not needed as primitives in the language for presenting the compositional semantics and logic in Chapter 3, but will be useful for bytecode optimizations presented in Chapter 5.)

A piece of code $c \in \mathbf{Code}$ is a finite set of labelled instructions, i.e., a set of pairs of a label and an instruction: $\mathbf{Code} =_{\mathrm{df}} \mathcal{P}_{\mathrm{fin}}(\mathbf{Label} \times \mathbf{Instr})$. A piece of code c is wellformed, if no label in it labels two different instructions, i.e., if $(\ell, \mathit{instr}), (\ell, \mathit{instr}') \in c$ imply $\mathit{instr} = \mathit{instr}'$. The domain of a piece of code is the set of labels in it: $\mathrm{dom}(c) =_{\mathrm{df}} \{\ell \mid (\ell, \mathit{instr}) \in c\}$. If $\ell \in \mathrm{dom}(c)$, we write c_ℓ for the unique instruction that ℓ labels in c.

A state for PUSH consists of a label ℓ, stack zs and store σ, which respectively record the program counter(pc) value, the content of the operand stack and the store at a moment: $\mathbf{State} =_{\mathrm{df}} \mathbf{Label} \times \mathbf{Stack} \times \mathbf{Store}$. A stack is a list whose elements can be both boolean or integer values: $\mathbf{Stack} =_{\mathrm{df}} (\mathbb{Z} + \mathbb{B})^*$. (We use the notation

X^* for lists over X, $[]$ for the empty list, $x :: xs$ for the list with head x and tail xs and $xs +\!\!+ ys$ for the concatenation of xs and ys.) Variables can only be of integer type and must always be defined: $\mathbf{Store} =_{\mathrm{df}} \mathbf{Var} \to \mathbb{Z}$.

If a language is low-level, its semantics is usually described in an operational form that is small-step. The small-step semantics of PUSH is formulated via a single-step reduction relation $- \vdash - \twoheadrightarrow - \subseteq \mathbf{State} \times \mathbf{Code} \times \mathbf{State}$ defined in Figure 2.3. The associated multi-step reduction relation $\twoheadrightarrow^*$ is its reflexive-transitive closure. It is immediate that $\twoheadrightarrow$ is deterministic, there is always at most one step possible. Additionally, we have the relation $- \vdash - \rightarrowtail\ \subseteq \mathbf{State} \times \mathbf{Code}$ denoting abnormal termination, also given in Figure 2.3. There is no explicit halt instructions, a program halts normally when the program counter exits the domain of the code.

Thus a state can be terminal for two reasons: (i) we have $\ell \notin \mathrm{dom}(c)$, which signifies normal termination (for which we write $c \vdash (\ell, zs, \sigma) \not\twoheadrightarrow$), or (ii) we have $\ell \in \mathrm{dom}(c)$ but have of wrong types or shortage of potential operands on the stack, which signifies abnormal termination (for which we write $c \vdash (\ell, zs, \sigma) \rightarrowtail$).

Lemma 1 (Determinacy of small-step semantics) *If $c \vdash (\ell, zs, \sigma) \twoheadrightarrow (\ell', zs', \sigma')$ and $c \vdash (\ell, zs, \sigma) \twoheadrightarrow (\ell'', zs'', \sigma'')$ then $(\ell'', zs'', \sigma'') = (\ell', zs', \sigma')$.*

Lemma 2 (Extension of the domain) *If $c_0 \subseteq c_1$, $\ell \in \mathrm{dom}(c_0)$ and $c_0 \vdash (\ell, zs, \sigma) \twoheadrightarrow^* (\ell', zs', \sigma')$ then $c_1 \vdash (\ell, zs, \sigma) \twoheadrightarrow^* (\ell', zs', \sigma')$.*

Lemma 3 (Extension of the domain 2) *If $c_0 \subseteq c_1$ and $\ell \in \mathrm{dom}(c_0)$ and $c_0 \vdash (\ell, zs, \sigma) \twoheadrightarrow^* (\ell', zs', \sigma') \rightarrowtail$ then $c_1 \vdash (\ell, zs, \sigma) \twoheadrightarrow^* (\ell', zs', \sigma') \rightarrowtail$.*

2.3 Program logics for Push

The Hoare logic of WHILE is very standard and can be found in most textbooks dealing with programming language semantics. For low-level languages, there is no such consensus and there is no canonical logic for simple low-level languages such as PUSH. The most popular approach involves a global context of label invariants, following the style of the type system for bytecode by Stata and Abadi [61], and can be found for example in Benton's [14] and Bannwart and Müller's [6] work. The logic we will use for presenting the low-level proof transformations in Chapter 5 is based on exactly this line of work.

When presenting a logic for a language in which errors can occur (type and stack underflow errors in our case), there are several design options. One question is whether the logic is error-ignoring or error-free. The difference between the two logics is in their validity statements. For the error-ignoring logic (also called *very partial correctness* logic) the validity statement says that if we start a program in a state satisfying the precondition, and terminate normally, the final state satisfies the postcondition. Such specifications are satisfied by both divergent and erroneous programs. For the error-free logic, validity additionally states that if we start a program in a state satisfying the precondition, the program *does not* finish erroneously.

$$\frac{P_\ell \models \forall z \in \mathbb{Z}, w \in (\mathbb{Z}+\mathbb{B})^*. (st = z :: w \Rightarrow P_{\ell+1}[w/st, z/x])}{P \vdash (\ell, \mathsf{store}\ x)} \text{ store}$$

$$\frac{P_\ell \models P_{\ell+1}[x :: st/st]}{P \vdash (\ell, \mathsf{load}\ x)} \text{ load} \qquad \frac{P_\ell \models P_{\ell+1}[n :: st/st]}{P \vdash (\ell, \mathsf{push}\ n)} \text{ push}$$

$$\frac{P_\ell \models \forall z_0, z_1 \in \mathbb{Z}, w \in (\mathbb{Z}+\mathbb{B})^*. (st = z_0 :: z_1 :: w \Rightarrow P_{\ell+1}[z_0 + z_1 : w/st])}{P \vdash (\ell, \mathsf{add})} \text{ add}$$

$$\ldots$$

$$\frac{P_\ell \models \forall z \in \mathbb{Z}+\mathbb{B}, w \in (\mathbb{Z}+\mathbb{B})^*. (st = z :: w \Rightarrow P_{\ell+1}[w/st])}{P \vdash (\ell, \mathsf{pop})} \text{ pop}$$

$$\frac{P_\ell \models \forall z \in \mathbb{Z}+\mathbb{B}, w \in (\mathbb{Z}+\mathbb{B})^*. st = z :: w \Rightarrow P_{\ell+1}[z :: z :: w/st]}{P \vdash (\ell, \mathsf{dup})} \text{ dup}$$

$$\frac{P_\ell \models P_m}{P \vdash (\ell, \mathsf{goto}\ m)} \text{ goto}$$

$$\frac{P_\ell \models \forall b \in \mathbb{B}, w \in (\mathbb{Z}+\mathbb{B})^*. st = b :: w \Rightarrow ((b \wedge P_{\ell+1}[w/st]) \vee (\neg b \wedge P_m[w/st]))}{P \vdash (\ell, \mathsf{gotoF}\ m)} \text{ gotoF}$$

$$\frac{\forall \ell \in \mathrm{dom}(c). P \vdash (\ell, c_\ell)}{P \vdash c} \text{ code}$$

Figure 2.4: Error-ignoring programming logic for PUSH

The two flavours of logic are given in 2.4 and Figures 2.5. There, P is a vector of assertions such that there is an assertion associated with each instruction of the program. We use P_ℓ to denote the assertion (more precisely, the precondition) at label ℓ. Assertions are formulae of the underlying logic, which as extra-arithmetic constants contains program variables **Var** and special constant st to refer to the stack. The logic judgement $P \vdash (\ell, instr)$ signifies that P is the context of invariants for the labelled instruction $(\ell, instr)$, independent of any possible code context in which this instruction may occur. The logic judgement $P \vdash c$ means that P is the context of invariants for a program c as a whole. The precise meaning of $P \vdash (\ell, instr)$ and $P \vdash c$ will be clear from the definition of soundness and completeness. We write $(zs, \sigma) \models_\alpha Q$ to express that an assertion Q holds in the structure on $\mathbb{Z}$ and $\mathbb{B}$ determined by (a) the standard meanings of the arithmetical and logical constants, function and predicate symbols and (b) a state σ and a stack st under an assignment α of the variables of the logical language (parameters).

The main difference between the two logics can be explained on the example of the store instruction. The error-free logic requires that the assertion associated with the store instruction guarantees that there is an integer on top of the the stack, so that the instruction could be executed normally. Additionally, the assertion has to imply the suitably modified assertion associated to the instruction following the store. In the error-ignoring logic on the other hand, the latter has to be guaranteed only if the stack holds an integer on top. If the stack is empty or holds the wrong

$$\frac{P_\ell \models \exists z \in \mathbb{Z}, w \in (\mathbb{Z}+\mathbb{B})^*.\, st = z :: w \wedge P_{\ell+1}[w/st, z/x]}{P \vdash (\ell, \mathsf{store}\ x)}\ \text{store}$$

$$\frac{P_\ell \models P_{\ell+1}[x :: st/st]}{P \vdash (\ell, \mathsf{load}\ x)}\ \text{load} \qquad \frac{P_\ell \models P_{\ell+1}[n :: st/st]}{P \vdash (\ell, \mathsf{push}\ n)}\ \text{push}$$

$$\frac{P_\ell \models \exists z_0, z_1 \in \mathbb{Z}, w \in (\mathbb{Z}+\mathbb{B})^*.\, st = z_0 :: z_1 :: w \wedge P_{\ell+1}[z_0 + z_1 : w/st]}{P \vdash (\ell, \mathsf{add})}\ \text{add}$$

$$\dots$$

$$\frac{P_\ell \models \exists z \in \mathbb{Z}+\mathbb{B}, w \in (\mathbb{Z}+\mathbb{B})^*.\, st = z :: w \wedge P_{\ell+1}[w/st]}{P \vdash (\ell, \mathsf{pop})}\ \text{pop}$$

$$\frac{P_\ell \models \exists z \in \mathbb{Z}+\mathbb{B}, w \in (\mathbb{Z}+\mathbb{B})^*.\, st = z :: w \wedge P_{\ell+1}[z :: z :: st/st]}{P \vdash (\ell, \mathsf{dup})}\ \text{dup}$$

$$\frac{P_\ell \models P_m}{P \vdash (\ell, \mathsf{goto}\ m)}\ \text{goto}$$

$$\frac{P_\ell \models \exists b \in \mathbb{B}, w \in (\mathbb{Z}+\mathbb{B})^*.\, st = b :: w \wedge ((b \wedge P_{\ell+1}[w/st]) \vee (\neg b \wedge P_m[w/st]))}{P \vdash (\ell, \mathsf{gotoF}\ m)}\ \text{gotoF}$$

$$\frac{\forall \ell \in \mathrm{dom}(c).\, P \vdash (\ell, c_\ell)}{P \vdash c}\ \text{code}$$

Figure 2.5: Error-free programming logic for PUSH

type on top, nothing has to be guaranteed, since a runtime error would occur.

Both logics are sound and complete in the usual sense, according to their validity notions.

Theorem 1 (Soundness of the error-ignoring programming logic) *If $P \vdash c$ and $(zs, \sigma) \models_\alpha P_\ell$ then for any (ℓ', zs', σ') such that $c \vdash (\ell, zs, \sigma) \twoheadrightarrow^* (\ell', zs', \sigma') \not\twoheadrightarrow$, $(zs', \sigma') \models_\alpha P_{\ell'}$.*

Theorem 2 (Completeness of error-ignoring programming logic) *If for any (ℓ, zs, σ) such that $(zs, \sigma) \models P_\ell$ it holds that for any (ℓ', zs', σ') such that $c \vdash (\ell, zs, \sigma) \twoheadrightarrow^* (\ell', zs', \sigma') \not\twoheadrightarrow$ we have $(zs', \sigma') \models P_{\ell'}$ then $P \vdash c$.*

Theorem 3 (Soundness of the error-free programming logic) *If $P \vdash c$ and $(zs, \sigma) \models_\alpha P_\ell$ then (i) for any (ℓ', zs', σ') such that $c \vdash (\ell, zs, \sigma) \twoheadrightarrow^* (\ell', zs', \sigma') \not\twoheadrightarrow$, $(zs', \sigma') \models_\alpha P_{\ell'}$ and (ii) there is no state (ℓ', zs', σ') such that $c \vdash (\ell, zs, \sigma) \twoheadrightarrow^* (\ell', zs', \sigma') \twoheadrightarrow\!\!|$.*

Theorem 4 (Completeness of the error-free programming logic) *If for any (ℓ, zs, σ) such that $(zs, \sigma) \models P_\ell$ it holds that (i) for any (ℓ', zs', σ') such that $c \vdash (\ell, zs, \sigma) \twoheadrightarrow^* (\ell', zs', \sigma') \not\twoheadrightarrow$ we have $(zs', \sigma') \models P_{\ell'}$ and (ii) there is no state (ℓ', zs', σ') such that $c \vdash (\ell, zs, \sigma) \twoheadrightarrow^* (\ell', zs', \sigma') \twoheadrightarrow\!\!|$ then $P \vdash c$.*

$$\frac{P_\ell \models \mathsf{shift}(P_{\ell+1})[st(0)/x]}{P \vdash (\ell, \mathsf{store}\ x)}\ \text{store} \qquad \frac{P_\ell \models \mathsf{unshift}(P_{\ell+1}[x/st(0)])}{P \vdash (\ell, \mathsf{load}\ x)}\ \text{load}$$

$$\frac{P_\ell \models \mathsf{unshift}(P_{\ell+1}[n/st(0)])}{P \vdash (\ell, \mathsf{push}\ n)}\ \text{push} \qquad \frac{P_\ell \models \mathsf{shift}(P_{\ell+1})[st(0) + st(1)/st(1)]}{P \vdash (\ell, \mathsf{add})}\ \text{add}$$

$$\dots$$

$$\frac{P_\ell \models \mathsf{shift}(P_{\ell+1})}{P \vdash (\ell, \mathsf{pop})}\ \text{pop} \qquad \frac{P_\ell \models \mathsf{unshift}(P_{\ell+1}[st(1)/st(0)])}{P \vdash (\ell, \mathsf{dup})}\ \text{dup}$$

$$\frac{P_\ell \models P_m}{P \vdash (\ell, \mathsf{goto}\ m)}\ \text{goto} \qquad \frac{P_\ell \models st(0) \wedge \mathsf{shift}(P_{\ell+1}) \vee \neg st(0) \wedge \mathsf{shift}(P_m)}{P \vdash (\ell, \mathsf{gotoF}\ m)}\ \text{gotoF}$$

$$\frac{\forall \ell \in \text{dom}(c).\, P \vdash (\ell, c_\ell)}{P \vdash c}\ \text{code}$$

Figure 2.6: Neutral programming logic for Push

There are in fact several ways of presenting logics that guarantee error freedom. For example Benton [14] uses a type system in parallel with the logic to rule out erroneous behavior. This means that nothing can be proven about untypable programs. This is a reasonable way to proceed (although for example Microsoft's CLR does give semantics to untypable code). The approach where error freedom is built directly into the logic (as in Figure 2.5) is stronger in the sense that certain properties of some untypable programs can also be proven, for example by showing that an ill-behaving branch is never entered. The compositional logic we will introduce in Chapter 3 will also be error-free.

For transformation of low-level proofs in Chapter 5, we will adopt a logic where the difference between error-free and error-ignoring is abstracted away, in fact the logic can be desugared both into the error-ignoring and error-free logic (however, the intended reading in Chapter 5 is error-ignoring). Its style follows the bytecode logics of Benton and Bannwart-Müller. In this notation, instead of having one extralogical constant st, we have multiple extralogical contstants $st(i)$ to refer to the i-th position in the stack, where i is a numeral and $st(0)$ is the top of the stack; using an arbitrary expression as the argument (such as $st(x+y)$) is not allowed. So for example to state that the third position in the stack holds a value 5, we have the assertion $st(2) = 5$.

The logic is given in Figure 2.6. We use operators shift and unshift where

$$\mathsf{shift}(Q) = Q[st(i+1)/st(i) \mid i \in \mathbb{N}]$$

and

$$\mathsf{unshift}(Q) = Q[st(i-1)/st(i) \mid i \in \mathbb{N} \wedge i > 0],$$

where Q is an assertion.

Using the $st(i)$ notation has the benefit of allowing us to talk about stack positions directly, without having to bring quantifiers into assertions to create bound variables for them. But there is also a drawback, namely it immediately begs the

question of how to interpret assertions referring to $st(i)$ when the length of the stack is less than or equal to i, for example $([], \sigma) \models st(0) = 1$. The interpretation depends on whether we consider the logic to be error-free or error-ignoring. In the first case, such an assertions should be false, in the second case it should be vacuously true.

We solve this issue, by stating that this logic with its problem of nondenoting terms can be desugared into the error-ignoring and error-free logics that are given in 2.4 and Figures 2.5 as needed. To translate the assertion Q containing extralogical constants $st(i), ..., st(n)$ for the error-ignoring logic, Q is rewritten as

$$\forall z_0, ...z_n, w.(st = z_0 :: ... :: z_n :: w \Rightarrow Q[z_0/st(0), .., z_n/st(n)]).$$

Similarly, for the error-free logic we would get

$$\exists z_0, ...z_n, w.(st = z_0 :: ... :: z_n :: w \wedge Q[z_0/st(0), .., z_n/st(n)]).$$

Proofs can also be translated.

Chapter 3

A compositional approach to low-level languages

Low-level languages are widely believed to be difficult to reason about because of their inherent non-modularity. The lack of modularity is attributed to low-level code being flat and to the presence of completely unrestricted jumps. The consequence of a language being non-modular is that it cannot have a compositional semantics or logic.

We show that the premise of non-modularity is untrue. While it is true that there is no explicit structure to low-level code in the way of high-level programs, there still is *some* structure: every piece of code is either a single labeled instruction, or a finite union of smaller pieces of code with non-overlapping support. As it turns out, this seemingly very banal structure actually provides a good enough phrase structure for defining a compositional semantics and logic for the language. The fundamental observation is that, differently from high-level statements, low-level pieces of code are multiple-entry and multiple-exit: any label could be jumped to and a piece of code could be exited from any jump instruction. The structure of finite unions is in fact natural and arises whenever code is produced by combining smaller pieces of code into larger ones, like it happens in a compiler.

In this chapter we introduce a compositional natural semantics and Hoare logic for the bytecode-like low-level language Push given in the previous chapter, and show how Hoare triples of While can be compiled down to the compositional logic together with the program.

3.1 Structured version of Push and its natural semantics

To overcome the non-compositionality problem of the semantics described above, some structure needs to be introduced into Push code. As mentioned, a useful structure to use for defining the semantics of a low-level language compositionally is that of finite unions of non-overlapping pieces of code. This is present in the code anyway, but it is ambiguous (any set is a finite union of disjoint sets in many ways)

and implicit, so one has to choose and make the choices explicit. Hence we define a corresponding structured version of PUSH, which we call SPUSH. Structured pieces of code $sc \in \mathbf{SCode}$ are defined by the following grammar

$$sc \quad ::= \quad (\ell, instr) \mid \mathbf{0} \mid sc_0 \oplus sc_1$$

which stipulates that a piece of code is either an empty piece of code, a single labelled instruction or a finite union of pieces of code. Instructions of SPUSH are the same as for PUSH, but we will skip the treatment of dup and pop in this chapter, since they do not offer anything for illustrating the compositional approach. They will be used in Chapter 5 for bytecode transformations. We define the domain $\mathrm{dom}(sc)$ of a piece of code sc to be the set of all labels in the code: $\mathrm{dom}(\mathbf{0}) = \emptyset$, $\mathrm{dom}((\ell, instr)) = \{l\}$, $\mathrm{dom}(sc_0 \oplus sc_1) = \mathrm{dom}(sc_0) \cup \mathrm{dom}(sc_1)$.

A piece of code is wellformed iff the labels of all of its instructions are different: a single instruction is always wellformed, $\mathbf{0}$ is wellformed and $sc_0 \oplus sc_1$ is wellformed iff both sc_0 and sc_1 are wellformed and $\mathrm{dom}(sc_0) \cap \mathrm{dom}(sc_1) = \emptyset$. Note that contiguity is not required for wellformedness, the domain of a piece of code does not have to be an interval.

The compositional semantic description we give for SPUSH is a natural (big-step) semantics. Since there is the possibility of abnormal terminations and we want to distinguish between non-terminations and abnormal terminations, we define two evaluation relations, $\succ\!-\!-\!\rightarrow, \succ\!-\!-\!\rightarrow\!\!\!\mid \; \subseteq \mathbf{State} \times \mathbf{SCode} \times \mathbf{State}$, one for normal evaluations, the other for abnormal terminating evaluations. Both relate possible initial states for evaluating a piece of code to the corresponding terminal states. The two relations are defined (mutually inductively) by the rules in Figure 3.1. Alternatively, we could have just one evaluation relation but indexed by a pair for distinguishing between the two flavors of termination.

The $\mathrm{load_{ns}}$ and $\mathrm{push_{ns}}$ rules should be self-explanatory. Both store x and add can potentially cause an error, therefore there are two rules for them, for normal and abnormal evaluation.

We have spelled out the rules for goto m and gotoF m instructions in two different ways: a recursive style (in square brackets) and a direct style. The two styles are equivalent, but we mostly comment the direct style. The recursive style could be seen as a formal explanation of the direct style. The key observation about the recursive rules should be that they work in conjunction with the $\mathrm{ood_{ns}}$ rule, which terminates the derivation if a jump is possible (otherwise, the recursive rule is applied again, infinitely in case of the goto rule). The issue is that, differently from other single-instruction pieces of code, a goto or gotoF instruction can loop back on itself. This happens when the current label and the target label coincide.

The side condition in the $\mathrm{goto_{ns}}$ rule states that a goto m instruction only terminates, if it does not loop directly back on itself. The $\mathrm{gotoF}^{\neq \mathrm{tt}}_{\mathrm{ns}}$ rule should be self-explanatory, however the gotoF m rules for the case there is a ff on the top of the stack should be explained. The complication here is that just like goto m, gotoF m can loop back on itself. Unlike goto m however, it cannot loop infinitely, since every

$$\frac{}{(\ell, zs, \sigma) \succ (\ell, \mathsf{load}\ x) \rightarrow (\ell+1, \sigma(x) :: zs, \sigma)}\ \text{load}_{\text{ns}}$$

$$\frac{n \in \mathbb{Z}}{(\ell, n :: zs, \sigma) \succ (\ell, \mathsf{store}\ x) \rightarrow (\ell+1, zs, \sigma[x \mapsto n])}\ \text{store}_{\text{ns}}$$

$$\frac{\forall n \in \mathbb{Z}, zs' \in (\mathbb{Z}+\mathbb{B})^*.\ zs \neq n :: zs'}{(\ell, zs, \sigma) \succ (\ell, \mathsf{store}\ x) \rightarrowtail (\ell, zs, \sigma)}\ \text{store}^{ab}_{\text{ns}}$$

$$\frac{}{(\ell, zs, \sigma) \succ (\ell, \mathsf{push}\ n) \rightarrow (\ell+1, n :: zs, \sigma)}\ \text{push}_{\text{ns}}$$

$$\frac{n_0, n_1 \in \mathbb{Z}}{(\ell, n_0 :: n_1 :: zs, \sigma) \succ (\ell, \mathsf{add}) \rightarrow (\ell+1, n_0 + n_1 :: zs, \sigma)}\ \text{add}_{\text{ns}}$$

$$\frac{\forall n_0, n_1 \in \mathbb{Z}, zs' \in (\mathbb{Z}+\mathbb{B})^*.\ zs \neq n_0 :: n_1 :: zs'}{(\ell, zs, \sigma) \succ (\ell, \mathsf{add}) \rightarrowtail (\ell, zs, \sigma)}\ \text{add}^{ab}_{\text{ns}}$$

$$\ldots$$

$$\left[\begin{array}{c} \dfrac{(m, zs, \sigma) \succ (\ell, \mathsf{goto}\ m) \rightarrow (\ell', zs', \sigma')}{(\ell, zs, \sigma) \succ (\ell, \mathsf{goto}\ m) \rightarrow (\ell', zs', \sigma')} \\ \dfrac{(m, zs, \sigma) \succ (\ell, \mathsf{goto}\ m) \rightarrowtail (\ell', zs', \sigma')}{(\ell, zs, \sigma) \succ (\ell, \mathsf{goto}\ m) \rightarrowtail (\ell', zs', \sigma')} \end{array}\right] \qquad \frac{m \neq \ell}{(\ell, zs, \sigma) \succ (\ell, \mathsf{goto}\ m) \rightarrow (m, zs, \sigma)}\ \text{goto}_{\text{ns}}$$

$$\left[\begin{array}{c} \dfrac{}{(\ell, \text{tt} :: zs, \sigma) \succ (\ell, \mathsf{gotoF}\ m) \rightarrow (\ell+1, zs, \sigma)} \\ \dfrac{(m, zs, \sigma) \succ (\ell, \mathsf{gotoF}\ m) \rightarrow (\ell', zs', \sigma')}{(\ell, \text{ff} :: zs, \sigma) \succ (\ell, \mathsf{gotoF}\ m) \rightarrow (\ell', zs', \sigma')} \\ \dfrac{(m, zs, \sigma) \succ (\ell, \mathsf{gotoF}\ m) \rightarrowtail (\ell', zs', \sigma')}{(\ell, \text{ff} :: zs, \sigma) \succ (\ell, \mathsf{gotoF}\ m) \rightarrowtail (\ell', zs', \sigma')} \\ \dfrac{\forall b \in \mathbb{B}, zs' \in (\mathbb{Z}+\mathbb{B})^*.\ zs \neq b :: zs'}{(\ell, zs, \sigma) \succ (\ell, \mathsf{gotoF}\ m) \rightarrowtail (\ell, zs, \sigma)} \end{array}\right]$$

$$\frac{m \neq \ell}{(\ell, \text{tt} :: zs, \sigma) \succ (\ell, \mathsf{gotoF}\ m) \rightarrow (\ell+1, zs, \sigma)}\ \text{gotoF}^{\neq\text{tt}}_{\text{ns}}$$

$$\frac{m \neq \ell}{(\ell, \text{ff} :: zs, \sigma) \succ (\ell, \mathsf{gotoF}\ m) \rightarrow (m, zs, \sigma)}\ \text{gotoF}^{\neq\text{ff}}_{\text{ns}}$$

$$\frac{m \neq \ell \quad \forall b \in \mathbb{B}, zs' \in (\mathbb{Z}+\mathbb{B})^*.\ zs \neq b :: zs'}{(\ell, zs, \sigma) \succ (\ell, \mathsf{gotoF}\ m) \rightarrowtail (\ell, zs, \sigma)}\ \text{gotoF}^{\neq ab}_{\text{ns}}$$

$$\frac{\mathit{ffs} \in \{\text{ff}\}^*}{(\ell, \mathit{ffs} \mathbin{+\!\!+} \text{tt} :: zs, \sigma) \succ (\ell, \mathsf{gotoF}\ \ell) \rightarrow (\ell+1, zs, \sigma)}\ \text{gotoF}^{=}_{\text{ns}}$$

$$\frac{\mathit{ffs} \in \{\text{ff}\}^* \quad \forall b \in \mathbb{B}, zs' \in (\mathbb{Z}+\mathbb{B})^*.\ zs \neq b :: zs'}{(\ell, \mathit{ffs} \mathbin{+\!\!+} zs, \sigma) \succ (\ell, \mathsf{gotoF}\ \ell) \rightarrowtail (\ell, zs, \sigma)}\ \text{gotoF}^{=ab}_{\text{ns}}$$

$$\frac{\ell \in \text{dom}(sc_i) \quad (\ell, zs, \sigma) \succ sc_i \rightarrow (\ell'', zs'', \sigma'') \quad (\ell'', zs'', \sigma'') \succ sc_0 \oplus sc_1 \rightarrow (\ell', zs', \sigma')}{(\ell, zs, \sigma) \succ sc_0 \oplus sc_1 \rightarrow (\ell', zs', \sigma')}\ \oplus_{\text{ns}}$$

$$\frac{\ell \in \text{dom}(sc_i) \quad (\ell, zs, \sigma) \succ sc_i \rightarrowtail (\ell', zs', \sigma')}{(\ell, zs, \sigma) \succ sc_0 \oplus sc_1 \rightarrowtail (\ell', zs', \sigma')}\ \oplus^{abn}_{\text{ns}}$$

$$\frac{\ell \in \text{dom}(sc_i) \quad (\ell, zs, \sigma) \succ sc_i \rightarrow (\ell'', zs'', \sigma'') \quad (\ell'', zs'', \sigma'') \succ sc_0 \oplus sc_1 \rightarrowtail (\ell', zs', \sigma')}{(\ell, zs, \sigma) \succ sc_0 \oplus sc_1 \rightarrowtail (\ell', zs', \sigma')}\ \oplus^{abl}_{\text{ns}}$$

$$\frac{\ell \notin \text{dom}(sc)}{(\ell, zs, \sigma) \succ sc \rightarrow (\ell, zs, \sigma)}\ \text{ood}_{\text{ns}}$$

Figure 3.1: Natural semantics rules of SPush

successful jump removes an element from the stack. Instead it can either exit the loop at some point (when it encounters a tt on top of the stack), or cause an error if it either encounters an integer on the stack or the stack runs empty. Therefore, two rules ($\text{gotoF}_{\text{ns}}^{=}$ and $\text{gotoF}_{\text{ns}}^{=ab}$) are needed for normal and abnormal behavior of $\mathsf{gotoF}\ m$ for the case when it loops back on itself. The rule $\text{gotoF}_{\text{ns}}^{\neq ab}$ covers the case when there is no boolean value at the top of the stack.

The rule $\oplus_{\text{ns}}$ says that, to evaluate the union $sc_0 \oplus sc_1$ starting from some state such that the pc is in the domain of sc_i, one first needs to evaluate sc_i, and then evaluate the whole union again, but starting from the new intermediate state reached. Finally, the ood_{ns} rule is needed to reflect the case where the reduction sequence is normally terminated because the pc has landed outside the domain of the code.

It is straightforward to show that the pc in the final state of a normally terminating evaluation of a code is outside its domain while the pc in the final state of an abnormally terminating evaluation is inside. Evaluation is deterministic in the sense that any piece of code terminates either normally or abnormally in a definite state, if it terminates at all.

Lemma 4 (Postlabels 1) *If* $(\ell, zs, \sigma) \succ sc \rightarrow (\ell', zs', \sigma')$*, then* $\ell' \notin \text{dom}(sc)$*.*

Lemma 5 (Postlabels 2) *If* $(\ell, zs, \sigma) \succ sc \rightarrowtail (\ell', zs', \sigma')$*, then* $\ell' \in \text{dom}(sc)$*.*

Lemma 6 (Determinacy 1) *If* $(\ell, zs, \sigma) \succ sc \rightarrow (\ell', zs', \sigma')$ *and* $(\ell, zs, \sigma) \succ sc \rightarrow (\ell'', zs'', \sigma'')$ *then* $(\ell', zs', \sigma') = (\ell'', zs'', \sigma'')$*.*

Lemma 7 (Determinacy 2) *If* $(\ell, zs, \sigma) \succ sc \rightarrowtail (\ell', zs', \sigma')$ *and* $(\ell, zs, \sigma) \succ sc \rightarrowtail (\ell'', zs'', \sigma'')$*, then* $(\ell', zs', \sigma') = (\ell'', zs'', \sigma'')$*.*

Every SPush piece of code can be mapped into a Push piece of code using a forgetful function $U \in \textbf{SCode} \rightarrow \textbf{Code}$, defined by $U((\ell, instr)) =_{\text{df}} \{(\ell, instr)\}$, $U(\mathbf{0}) =_{\text{df}} \emptyset$, $U(sc_0 \oplus sc_1) =_{\text{df}} U(sc_0) \cup U(sc_1)$. The compositional natural semantics of SPush agrees with the the standard small-step semantics of Push as given in Chapter 2 in the following technical sense.

Theorem 5 (Preservation of evaluations by U) *(i) If* $(\ell, zs, \sigma) \succ sc \rightarrow (\ell', zs', \sigma')$*, then* $U(sc) \vdash (\ell, zs, \sigma) \twoheadrightarrow^* (\ell', zs', \sigma') \not\twoheadrightarrow$*. (ii) If* $(\ell, zs, \sigma) \succ sc \rightarrowtail (\ell', zs', \sigma')$*, then* $U(sc) \vdash (\ell, zs, \sigma) \twoheadrightarrow^* (\ell', zs', \sigma') \rightarrowtail$*.*

Proof. By induction on the derivation of $(\ell, zs, \sigma) \succ sc \rightarrow (\ell', zs', \sigma')$ or $(\ell, zs, \sigma) \succ sc \rightarrowtail (\ell', zs', \sigma')$.

We look at normally terminating triples first and have the following cases:

- The derivation of $(\ell, zs, \sigma) \succ sc \rightarrow (\ell', zs', \sigma')$ is

$$\frac{}{(\ell, zs, \sigma) \succ (\ell, \mathsf{load}\ x) \rightarrow (\ell + 1, \sigma(x) :: zs, \sigma)}\ \text{load}_{\text{ns}}$$

  By rule load, we have $\{(\ell, \mathsf{load}\ x)\} \vdash (\ell, zs, \sigma) \twoheadrightarrow (\ell + 1, \sigma(x) :: zs, \sigma) \not\twoheadrightarrow$.

- The derivation of $(\ell, zs, \sigma) \succ sc\rightarrow (\ell', zs', \sigma')$ is

$$\frac{n \in \mathbb{Z}}{(\ell, n :: zs, \sigma) \succ (\ell, \textsf{store}\ x) \rightarrow (\ell+1, zs, \sigma[x \mapsto n])}\ \text{store}_{\text{ns}}$$

  By rule store, we have $\{(\ell, \textsf{store}\ x)\} \vdash (\ell, n :: zs, \sigma) \twoheadrightarrow (\ell+1, zs, \sigma[x \mapsto n]) \not\twoheadrightarrow$.

- The derivation of $(\ell, zs, \sigma) \succ sc\rightarrow (\ell', zs', \sigma')$ is

$$\frac{}{(\ell, zs, \sigma) \succ (\ell, \textsf{push}\ n) \rightarrow (\ell+1, n :: zs, \sigma)}\ \text{push}_{\text{ns}}$$

  By rule push, we have $\{(\ell, \textsf{push}\ n)\} \vdash (\ell, zs, \sigma) \twoheadrightarrow (\ell+1, n :: zs, \sigma) \not\twoheadrightarrow$.

- The derivation of $(\ell, zs, \sigma) \succ sc\rightarrow (\ell', zs', \sigma')$ is

$$\frac{n_0, n_1 \in \mathbb{Z}}{(\ell, n_0 :: n_1 :: zs, \sigma) \succ (\ell, \textsf{add}) \rightarrow (\ell+1, n_0 + n_1 :: zs, \sigma)}\ \text{add}_{\text{ns}}$$

  By rule add, we have $\{(\ell, \textsf{add})\} \vdash (\ell, n_0 :: n_1 :: zs, \sigma) \twoheadrightarrow (\ell+1, n_0 + n_1 :: zs, \sigma) \not\twoheadrightarrow$.

- The derivation of $(\ell, zs, \sigma) \succ sc\rightarrow (\ell', zs', \sigma')$ is

$$\frac{m \neq \ell}{(\ell, zs, \sigma) \succ (\ell, \textsf{goto}\ m) \rightarrow (m, zs, \sigma)}\ \text{goto}_{\text{ns}}$$

  By rule goto, we have $\{(\ell, \textsf{goto}\ m)\} \vdash (\ell, zs, \sigma) \twoheadrightarrow (m, zs, \sigma) \not\twoheadrightarrow$.

- The derivation of $(\ell, zs, \sigma) \succ sc\rightarrow (\ell', zs', \sigma')$ is

$$\frac{m \neq \ell}{(\ell, \text{tt} :: zs, \sigma) \succ (\ell, \textsf{gotoF}\ m) \rightarrow (\ell+1, zs, \sigma)}\ \text{gotoF}^{\neq\text{tt}}_{\text{ns}}$$

  By rule gotoF^{tt}, we have $\{(\ell, \textsf{gotoF}\ m)\} \vdash (\ell, \text{tt} :: zs, \sigma) \twoheadrightarrow (\ell+1, zs, \sigma) \not\twoheadrightarrow$.

- The derivation of $(\ell, zs, \sigma) \succ sc\rightarrow (\ell', zs', \sigma')$ is

$$\frac{m \neq \ell}{(\ell, \text{ff} :: zs, \sigma) \succ (\ell, \textsf{gotoF}\ m) \rightarrow (m, zs, \sigma)}\ \text{gotoF}^{\neq\text{ff}}_{\text{ns}}$$

  By rule gotoF^{ff}, we have $\{(\ell, \textsf{gotoF}\ m)\} \vdash (\ell, \text{ff} :: zs, \sigma) \twoheadrightarrow (m, zs, \sigma) \not\twoheadrightarrow$.

- The derivation of $(\ell, \sigma) \succ sc\rightarrow (\ell', \sigma')$ is

$$\frac{\mathit{ffs} \in \{\text{ff}\}^*}{(\ell, \mathit{ffs} +\!\!+ \text{tt} :: zs, \sigma) \succ (\ell, \textsf{gotoF}\ \ell) \rightarrow (\ell+1, zs, \sigma)}\ \text{gotoF}^{=}_{\text{ns}}$$

  If $\mathit{ffs} = []$, we have the derivation similar to the $\text{gotoF}^{\neq\text{tt}}_{\text{ns}}$ case. In the general case of $\mathit{ffs} = \mathit{ff}_0 :: ... :: \mathit{ff}_n :: []$, by rule gotoF^{ff}, we have the reduction sequence $\{(\ell, \textsf{gotoF}\ \ell)\} \vdash (\ell, \mathit{ff}_0 :: ... :: \mathit{ff}_n :: \text{tt} :: zs, \sigma) \twoheadrightarrow (\ell, \mathit{ff}_1 :: ... :: \mathit{ff}_n :: \text{tt} :: zs, \sigma) \twoheadrightarrow .. \twoheadrightarrow (\ell, \mathit{ff}_n :: \text{tt} :: zs, \sigma) \twoheadrightarrow (\ell, \text{tt} :: zs, \sigma) \twoheadrightarrow (\ell+1, zs, \sigma) \not\twoheadrightarrow$.

- The derivation of $(\ell, \sigma) \rightarrowtail sc \rightarrow (\ell', \sigma')$ is of the form

$$\frac{\ell \in \mathrm{dom}(sc_i) \quad (\ell, zs, \sigma) \rightarrowtail sc_i \rightarrow (\ell'', zs'', \sigma'') \quad (\ell'', zs'', \sigma'') \rightarrowtail sc_0 \oplus sc_1 \rightarrow (\ell', zs', \sigma')}{(\ell, zs, \sigma) \rightarrowtail sc_0 \oplus sc_1 \rightarrow (\ell', zs', \sigma')} \oplus_{\mathrm{ns}}$$

  where $i = 0$ or 1: By the induction hypothesis, we have $U(sc_i) \vdash (\ell, zs, \sigma) \twoheadrightarrow^* (\ell'', zs'', \sigma'') \not\twoheadrightarrow$ and $U(sc_0) \cup U(sc_1) \vdash (\ell'', zs'', \sigma'') \twoheadrightarrow^* (\ell', zs', \sigma') \not\twoheadrightarrow$. By Lemma 2, we have $U(sc_0) \cup U(sc_1) \vdash (\ell, zs, \sigma) \twoheadrightarrow^* (\ell'', zs'', \sigma'')$. Hence, $U(sc_0) \cup U(sc_1) \vdash (\ell, zs, \sigma) \twoheadrightarrow^* (\ell'', zs'', \sigma'') \twoheadrightarrow^* (\ell', zs'', \sigma') \not\twoheadrightarrow$.

- The derivation of $(\ell, \sigma) \rightarrowtail sc \rightarrow (\ell', \sigma')$ is

$$\frac{\ell \notin \mathrm{dom}(sc)}{(\ell, zs, \sigma) \rightarrowtail sc \rightarrow (\ell, zs, \sigma)} \mathrm{ood}_{\mathrm{ns}}$$

  We have $U(sc) \vdash (\ell, \sigma) \not\twoheadrightarrow$.

For the abnormally terminating triples, for primitive instructions it is easy to see the theorem holds, by inspecting the rule premises.

For composition, we have the following cases.

- The derivation has the form

$$\frac{\ell \in \mathrm{dom}(sc_i) \quad (\ell, zs, \sigma) \rightarrowtail sc_i \rightarrowtail\!\!| \, (\ell', zs', \sigma')}{(\ell, zs, \sigma) \rightarrowtail sc_0 \oplus sc_1 \rightarrowtail\!\!| \, (\ell', zs', \sigma')} \oplus_{\mathrm{ns}}^{abn}$$

  where $i = 0$ or 1. By the induction hypothesis, we have $U(sc_i) \vdash (\ell, zs, \sigma) \twoheadrightarrow^* (\ell', zs', \sigma') \rightarrow\!\!|$. Then by Lemma 3 we have $U(sc_0) \cup U(sc_1) \vdash (\ell, zs, \sigma) \twoheadrightarrow^* (\ell', zs', \sigma') \rightarrow\!\!|$.

- The derivation has the form

$$\frac{\ell \in \mathrm{dom}(sc_i) \quad (\ell, zs, \sigma) \rightarrowtail sc_i \rightarrow (\ell'', zs'', \sigma'') \quad (\ell'', zs'', \sigma'') \rightarrowtail sc_0 \oplus sc_1 \rightarrow\!\!| \, (\ell', zs', \sigma')}{(\ell, zs, \sigma) \rightarrowtail sc_0 \oplus sc_1 \rightarrow\!\!| \, (\ell', zs', \sigma')} \oplus_{\mathrm{ns}}^{abl}$$

  where $i = 0$ or 1. By the induction hypothesis (from the first half of the theorem), we have $U(sc_i) \vdash (\ell, zs, \sigma) \twoheadrightarrow^* (\ell'', zs'', \sigma'') \not\twoheadrightarrow$ and (from the second half of the theorem) $U(sc_0) \cup U(sc_1) \vdash (\ell'', zs'', \sigma'') \twoheadrightarrow^* (\ell', zs', \sigma') \rightarrow\!\!|$. By Lemma 2 we have $U(sc_0) \cup U(sc_1) \vdash (\ell, zs, \sigma) \twoheadrightarrow^* (\ell'', zs'', \sigma'')$, and therefore also $U(sc_0) \cup U(sc_1) \vdash (\ell, zs, \sigma) \twoheadrightarrow^* (\ell'', zs'', \sigma'') \twoheadrightarrow^* (\ell', zs', \sigma') \rightarrow\!\!|$.

□

Theorem 6 (Reflection of stuck reduction sequences by U) *(i) If $U(sc) \vdash (\ell_0, zs, \sigma) \twoheadrightarrow^* (\ell', zs', \sigma') \not\twoheadrightarrow$, then $(\ell_0, zs, \sigma) \rightarrowtail sc \rightarrow (\ell', zs', \sigma')$. (ii) If $U(sc) \vdash (\ell_0, zs, \sigma) \twoheadrightarrow^* (\ell', zs', \sigma') \rightarrow\!\!|$, then $(\ell_0, zs, \sigma) \rightarrowtail sc \rightarrow\!\!| \, (\ell', zs', \sigma')$*

Proof. By induction on the structure of sc and subordinate induction on the length of the reduction sequence.

We use structural induction on sc. We look at the normally terminating triples first. Assume we have a stuck reduction sequence $(\ell_0, zs_0, \sigma_0) \twoheadrightarrow \ldots \twoheadrightarrow (\ell_k, zs_k, \sigma_k) \not\twoheadrightarrow$ for $U(sc)$ $(k \geq 0)$ with the implications that $\ell_0, \ldots, \ell_{k-1} \in \mathrm{dom}(sc)$, $\ell_k \notin \mathrm{dom}(sc)$. There are the following cases.

- $sc = (\ell, \mathsf{load}\ x)$: If $\ell_0 = \ell$, then by Lemma 1 and rule load, the sequence can only be $(\ell, zs_0, \sigma_0) \twoheadrightarrow (\ell+1, \sigma(x) :: zs_0, \sigma_0) \not\twoheadrightarrow$. We have $(\ell, zs, \sigma) \succ (\ell, \mathsf{load}\ x) \rightarrow (\ell + 1, \sigma(x) :: zs, \sigma)$ by rule $\mathrm{load}_{\mathrm{ns}}$.

  If $\ell_0 \neq \ell$, then the sequence can only be $(\ell_0, zs_0, \sigma_0) \not\twoheadrightarrow$. We have $(\ell_0, zs, \sigma) \succ (\ell, \mathsf{load}\ x) \rightarrow (\ell_0, zs, \sigma)$ by rule $\mathrm{ood}_{\mathrm{ns}}$. The same will hold for the rest of the primitive instructions.

- $sc = (\ell, \mathsf{store}\ x)$: If $\ell_0 = \ell$, then by Lemma 1 and rule load, the sequence can only be $(\ell, n :: zs, \sigma) \twoheadrightarrow (\ell + 1, zs, \sigma[x \mapsto n]) \not\twoheadrightarrow$. We have $(\ell, n :: zs, \sigma) \succ (\ell, \mathsf{store}\ x) \rightarrow (\ell + 1, zs, \sigma[x \mapsto n])$ by rule $\mathrm{store}_{\mathrm{ns}}$.

  If $\ell_0 \neq \ell$, we have the derivation via the $\mathrm{ood}_{\mathrm{ns}}$ rule.

- $sc = (\ell, \mathsf{push}\ n)$: If $\ell_0 = \ell$, then by Lemma 1 and rule load, the sequence can only be $(\ell, zs_0, \sigma_0) \twoheadrightarrow (\ell+1, n :: zs_0, \sigma_0) \not\twoheadrightarrow$. We have $(\ell, zs, \sigma) \succ (\ell, \mathsf{push}\ n) \rightarrow (\ell + 1, n :: zs, \sigma)$ by rule $\mathrm{push}_{\mathrm{ns}}$.

  If $\ell_0 \neq \ell$, we have the derivation via the $\mathrm{ood}_{\mathrm{ns}}$ rule.

- $sc = (\ell, \mathsf{add})$: If $\ell_0 = \ell$, then by Lemma 1 and rule load, the sequence can only be $(\ell, n_0 :: n_1 :::: zs, \sigma) \twoheadrightarrow (\ell + 1, n_0 + n_1 :: zs, \sigma) \not\twoheadrightarrow$. We have $(\ell, n_0 :: n_1 :: zs, \sigma) \succ (\ell, \mathsf{add}) \rightarrow (\ell + 1, n_0 + n_1 :: zs, \sigma)$ by rule $\mathrm{add}_{\mathrm{ns}}$.

  If $\ell_0 \neq \ell$, we have the derivation via the $\mathrm{ood}_{\mathrm{ns}}$ rule.

- $sc = (\ell, \mathsf{goto}\ m)$: If $\ell_0 = \ell$ and $m \neq \ell$, then by Lemma 1 and rule goto, the sequence can only be $(\ell, zs_0, \sigma_0) \twoheadrightarrow (m, zs_0, \sigma_0) \not\twoheadrightarrow$ (we have $m \notin \{\ell\} = \mathrm{dom}(sc)$). We have $(\ell, zs_0, \sigma_0) \succ (\ell, \mathsf{goto}\ m) \rightarrow (m, zs_0, \sigma_0)$ by rule $\mathrm{goto}_{\mathrm{ns}}$.

  If $\ell_0 = \ell$ and $m = \ell$, then by Lemma 1 and rule goto a stuck reduction sequence is an impossibility (the only reduction sequence of (ℓ, zs_0, σ_0) is $(\ell, zs_0, \sigma_0) \twoheadrightarrow (\ell, zs_0, \sigma_0) \twoheadrightarrow \ldots$ and that never reaches a stuck state).

  If $\ell_0 \neq \ell$, we have the derivation via the $\mathrm{ood}_{\mathrm{ns}}$ rule.

- $sc = (\ell, \mathsf{gotoF}\ m)$: If $\ell_0 = \ell$ and $zs_0 = \mathrm{tt} :: w$, then by Lemma 1 and rule $\mathrm{gotoF}^{\mathrm{tt}}$ the sequence can only be $(\ell, \mathrm{tt} :: w, \sigma) \twoheadrightarrow (\ell + 1, w, \sigma) \not\twoheadrightarrow$. We have $(\ell, \mathrm{tt} :: w, \sigma) \succ (\ell, \mathsf{gotoF}\ m) \rightarrow (\ell + 1, w, \sigma)$ by rule $\mathrm{gotoF}^{\neq \mathrm{tt}}_{\mathrm{ns}}$.

  If $\ell_0 = \ell$, $zs_0 = \mathrm{ff} :: w$ and $m \neq \ell$, then by Lemma 1 and rule $\mathrm{gotoF}^{\mathrm{ff}}$ the sequence can only be $(\ell, \mathrm{ff} :: w, \sigma) \twoheadrightarrow (m, w, \sigma) \not\twoheadrightarrow$. We have $(\ell, \mathrm{tt} :: w, \sigma) \succ (\ell, \mathsf{gotoF}\ m) \rightarrow (\ell + 1, w, \sigma)$ by rule $\mathrm{gotoF}^{\neq \mathrm{ff}}_{\mathrm{ns}}$.

If $\ell_0 = \ell$, $zs_0 = \mathit{ffs} :: \mathrm{tt} :: w$ and $m \neq \ell$, where $\mathit{ffs} = \mathrm{ff}_0 :: ... :: \mathrm{ff}_n :: []$, then by Lemma 1 and rules $\mathrm{gotoF}^{\mathrm{ff}}$ and $\mathrm{gotoF}^{\mathrm{tt}}$ the sequence can only be $(\ell, \mathrm{ff}_0 :: ... :: \mathrm{ff}_n :: \mathrm{tt} :: w, \sigma) \twoheadrightarrow ... \twoheadrightarrow (\ell, \mathrm{ff}_n :: \mathrm{tt} :: w, \sigma) \twoheadrightarrow (\ell, \mathrm{tt} :: w, \sigma) \twoheadrightarrow (\ell+1, w, \sigma) \not\twoheadrightarrow$. We have $(\ell, \mathit{ffs} +\!\!+ \mathrm{tt} :: zs, \sigma) \succ (\ell, \mathsf{gotoF}\ \ell) \rightarrow (\ell+1, zs, \sigma)$ by rule $\mathrm{gotoF}^{=}_{\mathrm{ns}}$.

If $\ell_0 \neq \ell$, we have the derivation via the $\mathrm{ood}_{\mathrm{ns}}$ rule.

- $sc = \mathbf{0}$: The sequence can only be $(\ell_0, zs_0, \sigma_0) \not\twoheadrightarrow$ (since $\mathrm{dom}(c) = \emptyset$). We have $(\ell_0, zs_0, \sigma_0) \succ \mathbf{0} \rightarrow (\ell_0, zs_0, \sigma_0)$ by rule $\mathrm{ood}_{\mathrm{ns}}$.
- $sc = sc_0 \oplus sc_1$: We also invoke mathematical induction on k.

  If $\ell \in \mathrm{dom}(sc_i)$ for $i = 0$ or 1, then it must be that $k > 0$ and that $\ell_1, \ldots \ell_{k-1} \in \mathrm{dom}(sc_0) \cup \mathrm{dom}(sc_1)$ whereas $\ell_k \notin \mathrm{dom}(sc_0) \cup \mathrm{dom}(sc_1)$. Hence there must be a number j, $0 < j \leq k$, such that $\ell_1, \ldots \ell_{j-1} \in \mathrm{dom}(sc_i)$, but $\ell_j \notin \mathrm{dom}(sc_i)$. Our non-zero length stuck reduction sequence $(\ell_0, zs_0, \sigma_0) \twoheadrightarrow (\ell_1, zs_1, \sigma_1) \twoheadrightarrow^* (\ell_k, zs_k, \sigma_k) \not\twoheadrightarrow$ for $U(sc_0) \cup U(sc_1)$ splits into a non-zero length stuck reduction sequence $(\ell_0, zs_0, \sigma_0) \twoheadrightarrow (\ell_1, zs_1, \sigma_1) \twoheadrightarrow^* (\ell_j, zs_j, \sigma_j) \not\twoheadrightarrow$ for $U(sc_i)$ and a shorter than k stuck reduction sequence $(\ell_j, zs_j, \sigma_j) \twoheadrightarrow^* (\ell_k, zs_k, \sigma_k) \not\twoheadrightarrow$ for $U(sc_0) \cup U(sc_1)$. By the outer induction hypothesis, we have $(\ell_0, zs_0, \sigma_0) \succ sc_i \rightarrow (\ell_j, zs_j, \sigma_j)$. By the inner induction hypothesis, we have $(\ell_j, zs_j, \sigma_j) \succ sc_0 \oplus sc_1 \rightarrow (\ell_k, zs_k, \sigma_k)$. Hence we have $(\ell_0, zs_0, \sigma_0) \succ sc_0 \oplus sc_1 \rightarrow (\ell_k, zs_k, \sigma_k)$ by rule $\oplus_{\mathrm{ns}}$.

  If $\ell \notin \mathrm{dom}(sc_0)$ and $\ell \notin \mathrm{dom}(sc_1)$, then the sequence can only be $(\ell_0, zs_0, \sigma_0) \not\twoheadrightarrow$. We have $(\ell_0, zs_0, \sigma_0) \succ sc_0 \oplus sc_1 \rightarrow (\ell_0, zs_0, \sigma_0)$ by rule $\mathrm{ood}_{\mathrm{ns}}$.

For abnormally terminating triples assume we have a stuck reduction sequence $(\ell_0, zs_0, \sigma_0) \twoheadrightarrow \ldots \twoheadrightarrow (\ell_k, zs_k, \sigma_k) \rightarrow\!\!\mid$ for $U(sc)$ $(k \geq 0)$ and $\ell_0, \ldots, \ell_k \in \mathrm{dom}(sc)$. We omit the proof for primitive instruction and show it for composition.

- $sc = sc_0 \oplus sc_1$: We need to use mathematical induction on k.

  If $\ell \in \mathrm{dom}(sc_i)$ for $i = 0$ or 1, there are two possibilities.

  Either $(\ell_0, zs_0, \sigma_0) \twoheadrightarrow^* (\ell_k, zs_k, \sigma_k) \rightarrow\!\!\mid$, such that $\ell_k \in \mathrm{dom}(sc_i)$, or there must be a number j, $0 < j \leq k$, such that $\ell_0, \ldots \ell_{j-1} \in \mathrm{dom}(sc_i)$, but $\ell_j \notin \mathrm{dom}(sc_i)$.

  In the first case, we have $(\ell_0, zs_0, \sigma_0) \succ sc_i \rightarrow\!\!\mid (\ell_k, zs_k, \sigma_k)$ by the induction hypothesis and $(\ell_0, zs_0, \sigma_0) \succ sc_0 \oplus sc_1 \rightarrow\!\!\mid (\ell_k, zs_k, \sigma_k)$ via rule $\oplus^{abn}_{\mathrm{ns}}$

  In the second case, our non-zero length abnormally terminating reduction sequence $(\ell_0, zs_0, \sigma_0) \twoheadrightarrow (\ell_1, zs_1, \sigma_1) \twoheadrightarrow^* (\ell_k, zs_k, \sigma_k) \rightarrow\!\!\mid$ for $U(sc_0) \cup U(sc_1)$ splits into a non-zero length stuck reduction sequence $(\ell_0, zs_0, \sigma_0) \twoheadrightarrow (\ell_1, zs_1, \sigma_1) \twoheadrightarrow^* (\ell_j, zs_j, \sigma_j) \not\twoheadrightarrow$ for $U(sc_i)$ and a shorter than k stuck reduction sequence $(\ell_j, zs_j, \sigma_j) \twoheadrightarrow^* (\ell_k, zs_k, \sigma_k) \rightarrow\!\!\mid$ for $U(sc_0) \cup U(sc_1)$. By the outer induction hypothesis, we have $(\ell_0, zs_0, \sigma_0) \succ sc_i \rightarrow (\ell_j, zs_j, \sigma_j)$. By the inner induction hypothesis, we have $(\ell_j, zs_j, \sigma_j) \succ sc_0 \oplus sc_1 \rightarrow\!\!\mid (\ell_k, zs_k, \sigma_k)$. Hence we have $(\ell_0, zs_0, \sigma_0) \succ sc_0 \oplus sc_1 \rightarrow\!\!\mid (\ell_k, zs_k, \sigma_k)$ by rule $\oplus^{abl}_{\mathrm{ns}}$.

If $\ell \notin \mathrm{dom}(sc_0)$ and $\ell \notin \mathrm{dom}(sc_1)$, then there can be no derivation of $(\ell, zs, \sigma) \twoheadrightarrow^* (\ell', zs', \sigma')\nrightarrow$.

$\square$

From these theorems it is immediate that the SPush semantics of a structured version of a piece of Push code cannot depend on the way it is structured: if $U(sc) = U(sc')$, then sc and sc' have exactly the same evaluations (although with different derivations).

3.2 Hoare logic

The compositional natural semantics of SPush is a good basis for developing a compositional Hoare logic of it. Just as evaluations relate an initial and a terminal state, Hoare triples relate pre- and postconditions about states. Since a state contains a pc value and stack content, it must be possible to refer to these in assertions. In our logic, we have special individual constants pc and st to refer to them. Using the constant pc, we can make assertions about particular program points by constraining the state to correspond to a certain pc value. This allows us to make assertions only about program points through which the particular piece of code is entered or exited, thus eliminating the need for global contexts of invariants and making reasoning modular.

The logic we define is an error-free partial correctness logic: for a Hoare triple to be derivable, the postcondition must be satisfied by the terminal state of any normal evaluation and abnormal evaluations from the allowed initial states must be impossible. (We would get a more expressive partial correctness logic with triples with two postconditions, one for normal terminations, the other for abnormal terminations; in the case of a programming language with error-handling constructs, that approach is the only reasonable one, see, e.g., [58]. Our logic corresponds to the case where the abnormal postcondition is always $\bot$, so there is no need to ever spell it out. A different version where it is always $\top$ would correspond to error-ignoring partial correctness.)

The signature of the Hoare logic contains, as extra-arithmetical and extra-list constants, special individual constants pc, st and the program variables **Var**, to refer to the values of the program counter, stack and program variables in a state. The assertions $P, Q \in \mathbf{Assn}$ are formulae over that signature in an ambient logical language containing the signature of arithmetic and lists of integers and booleans. We use the notation $Q[t_0, .., t_n/x_0, .., x_n]$ to denote that every occurrence of x_i in Q has been replaced with t_i. The derivable Hoare triples $\{\} - \{\} \subseteq \mathbf{Assn} \times \mathbf{SCode} \times \mathbf{Assn}$ are defined inductively by the rules in Figure 3.2.

The extra disjunct $pc \neq \ell \wedge Q$ in the rules for primitive instructions is required because of the semantic rule $\mathrm{ood}_{\mathrm{ns}}$: if we evaluate the instruction starting from outside the domain of the instruction (i.e. $pc \neq \ell$), we have immediately terminated and have hence remained in the same state, therefore any assertion holding before

$$\frac{}{\{(pc = \ell \wedge Q[\ell+1, x :: st/pc, st]) \vee (pc \neq \ell \wedge Q)\}\ (\ell, \mathsf{load}\ x)\ \{Q\}}\ \mathrm{load_{hoa}}$$

$$\frac{}{\left\{\begin{array}{ll} & (pc = \ell \wedge \exists z \in \mathbb{Z}, w \in (\mathbb{Z}+\mathbb{B})^*.\ st = z :: w \wedge Q[\ell+1, w, z/pc, st, x]) \\ \vee & (pc \neq \ell \wedge Q) \end{array}\right\}\ (\ell, \mathsf{store}\ x)\ \{\ Q\ \}}\ \mathrm{store_{hoa}}$$

$$\frac{}{\{(pc = \ell \wedge Q[\ell+1, n :: st/pc, st]) \vee (pc \neq \ell \wedge Q)\}\ (\ell, \mathsf{push}\ n)\ \{Q\}}\ \mathrm{push_{hoa}}$$

$$\frac{}{\left\{\begin{array}{ll} & (pc = \ell \wedge \exists z_0, z_1 \in \mathbb{Z}, w \in (\mathbb{Z}+\mathbb{B})^*.\ st = z_0 :: z_1 :: w \\ & \quad \wedge Q[\ell+1, z_0 + z_1 :: w/pc, st]) \\ \vee & (pc \neq \ell \wedge Q) \end{array}\right\}\ (\ell, \mathsf{add})\ \{\ Q\ \}}\ \mathrm{add_{hoa}}$$

$$\ldots$$

$$\frac{}{\left\{\begin{array}{ll} & (pc = \ell \wedge ((m \neq \ell \wedge Q[m/pc]) \vee m = \ell)) \\ \vee & (pc \neq \ell \wedge Q) \end{array}\right\}\ (\ell, \mathsf{goto}\ m)\ \{\ Q\ \}}\ \mathrm{goto_{hoa}}$$

$$\frac{}{\left\{\begin{array}{llll} & (pc = \ell & \wedge((m \neq \ell & \wedge((\exists w \in (\mathbb{Z}+\mathbb{B})^*.\ st = \mathrm{tt} :: w \\ & & & \quad \wedge Q[\ell+1, w/pc, st]) \\ & & & \vee(\exists w \in (\mathbb{Z}+\mathbb{B})^*.\ st = \mathrm{ff} :: w \\ & & & \quad \wedge Q[m, w/pc, st]))) \\ & & \vee(m = \ell & \wedge \exists \mathit{ffs} \in \{\mathrm{ff}\}^*, w \in (\mathbb{Z}+\mathbb{B})^*. st = \mathit{ffs} +\!\!+ \mathrm{tt} :: w \\ & & & \quad \wedge Q[\ell+1, w/pc, st]))) \\ \vee & (pc \neq \ell & \wedge Q) & \end{array}\right\}\ (\ell, \mathsf{gotoF}\ m)\ \{\ Q\ \}}\ \mathrm{gotoF_{hoa}}$$

$$\frac{}{\{P\}\ \mathbf{0}\ \{P\}}\ \mathbf{0}_{\mathrm{hoa}} \qquad \frac{\{pc \in \mathrm{dom}(sc_0) \wedge P\}\ sc_0\ \{P\} \quad \{pc \in \mathrm{dom}(sc_1) \wedge P\}\ sc_1\ \{P\}}{\{P\}\ sc_0 \oplus sc_1\ \{pc \notin \mathrm{dom}(sc_0) \wedge pc \notin \mathrm{dom}(sc_1) \wedge P\}}\ \oplus_{\mathrm{hoa}}$$

$$\frac{P \models P' \quad \{P'\}\ sc\ \{Q'\} \quad Q' \models Q}{\{P\}\ sc\ \{Q\}}\ \mathrm{conseq_{hoa}}$$

Figure 3.2: Hoare rules of SPush

evaluating the instruction will also hold after. The disjunct $m = \ell$ in the rule for goto m accounts for the case when goto m loops back on itself. We have a similar case with the gotoF m rule, but here the situation is more subtle. As explained in Section 3.1, when gotoF m loops back on itself, it can either exit normally to the next instruction (in case there is some number of ffs on the stack, followed by a tt), or raise an error. The disjunct $m = \ell \wedge ..$ accounts for that case.

The rule for unions can be seen as mix of the while and sequence rules of the Hoare logic of While: if, evaluating either sc_0 or sc_1 starting from a state that satisfies P and has the pc value in the domain of sc_0 resp. sc_1, we end in a state satisfying P, then, after evaluating their union $sc_0 \oplus sc_1$ starting from a state satisfying P, we are guaranteed to be in a state satisfying P. Furthermore, we know that we are then outside the domains of both sc_0 and sc_1. The rule of consequence is the same as in the standard Hoare logic. Note that we have circumvented the inevitable *incompleteness* of any axiomatization of logics containing arithmetic by invoking semantic entailment instead of deducibility in the premises of the conseq rule.

The Hoare logic is sound and complete.

Theorem 7 (Soundness of Hoare logic) *If* $\{P\}\ sc\ \{Q\}$ *and* $(\ell_0, zs_0, \sigma_0) \models_\alpha P$, *then (i) for any* (ℓ', zs', σ') *such that* $(\ell_0, zs_0, \sigma_0) \succ\!sc\!\rightarrow (\ell', zs', \sigma')$, *we have* $(\ell', zs', \sigma') \models_\alpha Q$, *and (ii) there is no* (ℓ', zs', σ') *such that* $(\ell_0, zs_0, \sigma_0) \succ\!sc\!\rightarrow\!\!\!\mid (\ell', zs', \sigma')$.

Proof. By induction on the derivation of $\{P\}\, sc\, \{Q\}$. We have the following cases.

- The derivation of $\{P\}\, sc\, \{Q\}$ is

$$\frac{}{\{(pc = \ell \wedge Q[\ell+1, x :: st/pc, st]) \vee (pc \neq \ell \wedge Q)\}\, (\ell, \mathsf{load}\ x)\, \{Q\}}\ \text{load}_{\text{hoa}}$$

  Suppose $(\ell_0, zs_0, \sigma_0) \models_\alpha (pc = \ell \wedge Q[\ell+1, x :: st/pc, st]) \vee (pc \neq \ell \wedge Q)$ and $(\ell_0, zs_0, \sigma_0) \succ\!(\ell, \mathsf{load}\ x)\!\rightarrow (\ell', zs', \sigma')$ for some $\ell_0, zs_0, \sigma_0, \ell', zs', \sigma', \alpha$.

  If $\ell_0 = \ell$, then $(\ell_0, zs_0, \sigma_0) \models_\alpha (pc = \ell \wedge Q[\ell+1, x :: st/pc, st])$. By Lemma 6 and load_{ns} rule, $(\ell', zs', \sigma') = (\ell+1, \sigma_0(x) :: zs_0, \sigma_0)$. Hence, $(\ell', zs', \sigma') \models_\alpha Q$.

  If $\ell_0 \neq \ell$, then $(\ell_0, zs_0, \sigma_0) \models_\alpha (pc \neq \ell \wedge Q)$. From Lemma 6 and ood_{ns} rule, we get that $(\ell_0, zs_0, \sigma_0) = (\ell', zs', \sigma')$, so $(\ell', zs', \sigma') \models_\alpha Q$ trivially holds. There is no possibility for abnormal termination for $\mathsf{load}\ x$. The same will hold for other primitive instructions for the case where $\ell_0 \neq \ell$.

- The derivation of $\{P\}\, sc\, \{Q\}$ is

$$\frac{}{\left\{\begin{array}{ll} & (pc = \ell \wedge \exists z \in \mathbb{Z}, w \in (\mathbb{Z}+\mathbb{B})^*.\, st = z :: w \\ & \quad \wedge Q[\ell+1, w, z/pc, st, x]) \\ \vee & (pc \neq \ell \wedge Q) \end{array}\right\} (\ell, \mathsf{store}\ x)\ \{\ Q\ \}}\ \text{store}_{\text{hoa}}$$

  Suppose $(\ell_0, zs_0, \sigma_0) \models_\alpha (pc = \ell \wedge \exists z \in \mathbb{Z}, w \in (\mathbb{Z}+\mathbb{B})^*.\, st = z :: w \wedge Q[\ell + 1, w, z/pc, st, x]) \vee (pc \neq \ell \wedge Q)$ and $(\ell_0, zs_0, \sigma_0) \succ\!(\ell, \mathsf{store}\ x)\!\rightarrow (\ell', zs', \sigma')$ for some $\ell_0, zs_0, \sigma_0, \ell', zs', \sigma', \alpha$.

  If $\ell_0 = \ell$, then $(\ell_0, zs_0, \sigma_0) \models_\alpha (pc = \ell \wedge \exists z \in \mathbb{Z}, w \in (\mathbb{Z}+\mathbb{B})^*.\, st = z :: w \wedge Q[\ell+1, w, z/pc, st, x])$. Since $\exists z \in \mathbb{Z}, w \in (\mathbb{Z}+\mathbb{B})^*.\, st = z :: w$ holds, the abnormal rule for $\mathsf{store}\ x$ is not applicable and there has to exist an n and zs_0' s.t. $zs_0 = n :: zs_0'$. By Lemma 6 and the store_{ns} rule, $(\ell', zs', \sigma') = (\ell_0 + 1, zs_0', \sigma_0[x \mapsto n])$. Hence, $(\ell', zs', \sigma') \models_\alpha Q$.

  If $\ell_0 \neq \ell$, then $(\ell_0, zs_0, \sigma_0) \models_\alpha (pc \neq \ell \wedge Q)$. From Lemma 6 and the ood_{ns} rule, we get that $(\ell_0, zs_0, \sigma_0) = (\ell', zs', \sigma')$, so $(\ell', zs', \sigma') \models_\alpha Q$ trivially holds.

- The derivation of $\{P\}\, sc\, \{Q\}$ is

$$\frac{}{\{(pc = \ell \wedge Q[\ell+1, n :: st/pc, st]) \vee (pc \neq \ell \wedge Q)\}\, (\ell, \mathsf{push}\ n)\, \{Q\}}\ \text{push}_{\text{hoa}}$$

  Suppose $(\ell_0, zs_0, \sigma_0) \models_\alpha (pc = \ell \wedge Q[\ell+1, n :: st/pc, st]) \vee (pc \neq \ell \wedge Q)$ and $(\ell_0, zs_0, \sigma_0) \succ\!(\ell, \mathsf{push}\ n)\!\rightarrow (\ell', zs', \sigma')$ for some $\ell_0, zs_0, \sigma_0, \ell', zs', \sigma', \alpha$.

  If $\ell_0 = \ell$, then $(\ell_0, zs_0, \sigma_0) \models_\alpha (pc = \ell \wedge Q[\ell+1, n :: st/pc, st])$. By Lemma 6 and the push_{ns} rule, $(\ell', zs', \sigma') = (\ell_0+1, n :: zs, \sigma_0)$. Hence, $(\ell', zs', \sigma') \models_\alpha Q$.

  If $\ell_0 \neq \ell$, then $(\ell_0, zs_0, \sigma_0) \models_\alpha (pc \neq \ell \wedge Q)$. From Lemma 6 and the ood_{ns} rule, we get that $(\ell_0, zs_0, \sigma_0) = (\ell', zs', \sigma')$, so $(\ell', zs', \sigma') \models_\alpha Q$ trivially holds. There is no possibility for abnormal termination for $\mathsf{push}\ x$.

- The derivation of $\{P\}\, sc\, \{Q\}$ is

$$\dfrac{}{\left\{\begin{array}{ll} & (pc = \ell \wedge \exists z_0, z_1 \in \mathbb{Z}, w \in (\mathbb{Z}+\mathbb{B})^*.\, st = z_0 :: z_1 :: w \\ & \wedge Q[\ell+1, z_0 + z_1 :: w/pc, st]) \\ \vee & (pc \neq \ell \wedge Q) \end{array}\right\} (\ell, \mathsf{add})\ \{\ Q\ \}}\ \text{add}_{\text{hoa}}$$

  Suppose $(\ell_0, zs_0, \sigma_0) \models_\alpha (pc = \ell \wedge \exists z_0, z_1 \in \mathbb{Z}, w \in (\mathbb{Z}+\mathbb{B})^*.\, st = z_0 :: z_1 :: w \wedge Q[\ell+1, z_0+z_1 :: w/pc, st]) \vee (pc \neq \ell \wedge Q)$ and $(\ell_0, zs_0, \sigma_0) \succ\!(\ell, \mathsf{add})\!\rightarrow (\ell', zs', \sigma')$ for some $\ell_0, zs_0, \sigma_0, \ell', zs', \sigma', \alpha$.

  If $\ell_0 = \ell$, then $(\ell_0, zs_0, \sigma_0) \models_\alpha (pc = \ell \wedge \exists z_0, z_1 \in \mathbb{Z}, w \in (\mathbb{Z}+\mathbb{B})^*.\, st = z_0 :: z_1 :: w \wedge Q[\ell+1, z_0+z_1 :: w/pc, st])$. Since $\exists z_0, z_1 \in \mathbb{Z}, w \in (\mathbb{Z}+\mathbb{B})^*.\, st = z_0 :: z_1 :: w$ holds, the abnormal rule for add is not applicable. There have to exist n_1, n_2 and zs'_0 such that $zs_0 = n_1 :: n_2 :: zs'_0$. By Lemma 6 and the add_{ns} rule, $(\ell', zs', \sigma') = (\ell+1, n_0+n_1 :: zs'_0, \sigma_0)$. Hence, $(\ell', zs', \sigma') \models_\alpha Q$.

  If $\ell_0 \neq \ell$, then $(\ell_0, zs_0, \sigma_0) \models_\alpha (pc \neq \ell \wedge Q)$. From Lemma 6 and ood_{ns} rule, we get that $(\ell_0, zs_0, \sigma_0) = (\ell', zs', \sigma')$, so $(\ell', zs', \sigma') \models_\alpha Q$ trivially holds.

- The derivation of $\{P\}\, sc\, \{Q\}$ is

$$\dfrac{}{\left\{\begin{array}{ll} & (pc = \ell \wedge ((m \neq \ell \wedge Q[m/pc]) \vee m = \ell)) \\ \vee & (pc \neq \ell \wedge Q) \end{array}\right\} (\ell, \mathsf{goto}\ m)\ \{\ Q\ \}}\ \text{goto}_{\text{hoa}}$$

  Suppose $(\ell_0, zs_0, \sigma_0) \models_\alpha (pc = \ell \wedge ((m \neq \ell \wedge Q[m/pc]) \vee m = \ell)) \vee (pc \neq \ell \wedge Q)$ and $(\ell_0, zs_0, \sigma_0) \succ\!(\ell, \mathsf{goto}\ m)\!\rightarrow (\ell', zs', \sigma')$ for some $\ell_0, zs_0, \sigma_0, \ell', zs', \sigma', \alpha$.

  If $\ell_0 = \ell$, then $(pc = \ell \wedge ((m \neq \ell \wedge Q[m/pc]) \vee m = \ell))$. By Lemma 6 and the goto rule, we have that $m \neq \ell_0$ and $(\ell', zs', \sigma') = (m, zs, \sigma)$. Hence, $(\ell', zs', \sigma') \models_\alpha Q$.

  If $\ell_0 \neq \ell$, then $(\ell_0, zs_0, \sigma_0) \models_\alpha (pc \neq \ell \wedge Q)$. From Lemma 6 and ood_{ns} rule, we get that $(\ell_0, zs_0, \sigma_0) = (\ell', zs', \sigma')$, so $(\ell', zs', \sigma') \models_\alpha Q$ trivially holds.

- The derivation of $\{P\}\, sc\, \{Q\}$ is

$$\dfrac{}{\left\{\begin{array}{llll} & (pc = \ell & \wedge((m \neq \ell & \wedge((\exists w \in (\mathbb{Z}+\mathbb{B})^*.\, st = \mathrm{tt} :: w \\ & & & \quad \wedge Q[\ell+1, w/pc, st]) \\ & & & \vee(\exists w \in (\mathbb{Z}+\mathbb{B})^*.\, st = \mathrm{ff} :: w \\ & & & \quad \wedge Q[m, w/pc, st]))) \\ & & \vee(m = \ell & \wedge \exists \mathit{ffs} \in \{\mathrm{ff}\}^*, w \in (\mathbb{Z}+\mathbb{B})^*. st = \mathit{ffs} +\!\!+ \mathrm{tt} :: w \\ & & & \quad \wedge Q[\ell+1, w/pc, st]))) \\ \vee & (pc \neq \ell & \wedge Q) & \end{array}\right\} (\ell, \mathsf{gotoF}\ m)\ \{\ Q\ \}}\ \text{gotoF}_{\text{hoa}}$$

  Suppose $(\ell_0, zs_0, \sigma_0) \models_\alpha (pc = \ell \wedge ((m \neq \ell \wedge ((\exists w \in (\mathbb{Z}+\mathbb{B})^*.\, st = \mathrm{tt} :: w \wedge Q[\ell+1, w/pc, st]) \vee (\exists w \in (\mathbb{Z}+\mathbb{B})^*.\, st = \mathrm{ff} :: w \wedge Q[m, w/pc, st]))) \vee (m = \ell \wedge \exists \mathit{ffs} \in \{\mathrm{ff}\}^*, w \in (\mathbb{Z}+\mathbb{B})^*. st = \mathit{ffs} +\!\!+ \mathrm{tt} :: w \wedge Q[\ell+1, w/pc, st]))) \vee (pc \neq \ell \wedge Q)$ and $(\ell_0, zs_0, \sigma_0) \succ\!(\ell, \mathsf{gotoF}\ m)\!\rightarrow (\ell', zs', \sigma')$ for some $\ell_0, zs_0, \sigma_0, \ell', zs', \sigma', \alpha$.

  We get three cases.

  If $\ell_0 = \ell$, $m \neq \ell$ and $(\ell_0, zs_0, \sigma_0) \models_\alpha \exists w \in (\mathbb{Z}+\mathbb{B})^*.\, st = \mathrm{tt} :: w$, then $(\ell_0, zs_0, \sigma_0) \models_\alpha Q[\ell+1, w/pc, st]$. There has to exist a zs'_0 such that $zs_0 =$

$\mathrm{tt} :: zs'_0$. By Lemma 6 and the $\mathrm{gotoF}^{\neq \mathrm{tt}}_{\mathrm{ns}}$ rule, we have that $(\ell', zs', \sigma') = (\ell_0 + 1, zs'_0, \sigma_0)$. Hence, $(\ell', zs', \sigma') \models_\alpha Q$.

If $\ell_0 = \ell$, $m \neq \ell$ and $(\ell_0, zs_0, \sigma_0) \models_\alpha \exists w \in (\mathbb{Z} + \mathbb{B})^*.\, st = \mathrm{ff} :: w$, then $(\ell_0, zs_0, \sigma_0) \models_\alpha Q[m, w/pc, st]$. There has to exist a zs'_0 such that $zs_0 = \mathrm{ff} :: zs'_0$. By Lemma 6 and $\mathrm{gotoF}^{\neq \mathrm{ff}}_{\mathrm{ns}}$ rule, we have that $(\ell', zs', \sigma') = (m, zs'_0, \sigma_0)$. Hence, $(\ell', zs', \sigma') \models_\alpha Q$.

In both cases, there is no possibility of abnormal termination.

If $\ell_0 = \ell$, $m = \ell$ and $(\ell_0, zs_0, \sigma_0) \models_\alpha \exists \mathit{ffs} \in \{\mathrm{ff}\}^*, w \in (\mathbb{Z} + \mathbb{B})^*. st = \mathit{ffs} +\!\!+ \mathrm{tt} :: w$, then $(\ell_0, zs_0, \sigma_0) \models_\alpha Q[\ell + 1, w/pc, st]$. There has to exist a zs'_0 such that $zs_0 = \mathrm{ff} ... \mathrm{ff} :: \mathrm{tt} :: zs'_0$. By Lemma 6 and $\mathrm{gotoF}^{=}_{\mathrm{ns}}$ rules, we have that $(\ell', zs', \sigma') = (\ell_0 + 1, zs'_0, \sigma_0)$. Hence $(\ell', zs', \sigma') \models_\alpha Q$. Again, no application of abnormal termination rule is possible.

If $\ell_0 \neq \ell$, then $(\ell_0, zs_0, \sigma_0) \models_\alpha (pc \neq \ell \wedge Q)$. From Lemma 6 and $\mathrm{ood}_{\mathrm{ns}}$ rule, we get that $(\ell_0, zs_0, \sigma_0) = (\ell', zs', \sigma')$, so $(\ell', zs', \sigma') \models_\alpha Q$ trivially holds.

- The derivation of $\{P\}\, sc\, \{Q\}$ is

$$\frac{}{\{P\}\, \mathbf{0}\, \{P\}}\ \mathbf{0}_{\mathrm{hoa}}$$

Suppose $(\ell_0, zs_0, \sigma_0) \models_\alpha Q$ and and $(\ell_0, zs_0, \sigma_0) \succ\!\mathbf{0}\!\rightarrow (\ell', zs', \sigma')$ for some $\ell, zs, \sigma, \ell', zs', \sigma', \alpha$. Then by Lemma 6 and rule $\mathrm{ood}_{\mathrm{ns}}$, $(\ell_0, zs_0, \sigma_0) = (\ell', zs', \sigma')$, and $(\ell', zs', \sigma') \models_\alpha Q$ trivially holds. Abnormal termination is not possible.

- The derivation of $\{P\}\, sc\, \{Q\}$ is

$$\frac{\begin{array}{cc} \vdots & \vdots \\ \{pc \in \mathrm{dom}(sc_0) \wedge P\}\, sc_0\, \{P\} & \{pc \in \mathrm{dom}(sc_1) \wedge P\}\, sc_1\, \{P\} \end{array}}{\{P\}\, sc_0 \oplus sc_1\, \{pc \notin \mathrm{dom}(sc_0) \wedge pc \notin \mathrm{dom}(sc_1) \wedge P\}}\ \oplus_{\mathrm{hoa}}$$

Suppose $(\ell_0, zs_0, \sigma_0) \models_\alpha P$ and $(\ell_0, zs_0, \sigma_0) \succ\! sc_0 \oplus sc_1 \!\rightarrow (\ell', zs', \sigma')$ for some $\ell_0, zs_0, \sigma_0, \ell', zs', \sigma'$ and α. We invoke structural induction on the derivation of $(\ell_0, zs_0, \sigma_0) \succ\! sc_0 \oplus sc_1 \!\rightarrow (\ell', zs', \sigma')$.

If $\ell \in \mathrm{dom}(sc_i)$, then by Lemma 6, it must be that the last inference of the derivation of $(\ell_0, zs_0, \sigma_0) \succ\! sc_0 \oplus sc_1 \!\rightarrow (\ell', zs', \sigma')$ is an application of rule $\oplus^i_{\mathrm{ns}}$ to $(\ell_0, zs_0, \sigma_0) \succ\! sc_i \!\rightarrow (\ell'', zs'', \sigma'')$ and $(\ell'', zs'', \sigma'') \succ\! sc_0 \oplus sc_1 \!\rightarrow (\ell', zs', \sigma')$ for some ℓ'', zs'', σ''. We have $(\ell_0, zs_0, \sigma_0) \models_\alpha pc \in \mathrm{dom}(sc_i) \wedge P$, from where by the outer induction hypothesis $(\ell'', zs'', \sigma'') \models_\alpha P$ and further by the inner induction hypothesis $(\ell', zs', \sigma') \models_\alpha pc \notin \mathrm{dom}(sc_0) \wedge pc \notin \mathrm{dom}(sc_1) \wedge P$.

If $\ell \notin \mathrm{dom}(sc_0)$ and $\ell \notin \mathrm{dom}(sc_1)$, then by Lemma 6 and rule $\mathrm{ood}_{\mathrm{ns}}$, we get $(\ell', zs', \sigma') = (\ell_0, zs_0, \sigma_0)$, from where it follows that $(\ell', zs', \sigma') \models_\alpha pc \notin \mathrm{dom}(sc_0) \wedge pc \notin \mathrm{dom}(sc_1) \wedge P$.

- The derivation of $\{P\}\, sc\, \{Q\}$ is

$$\dfrac{P \models P' \quad \begin{array}{c}\vdots\\ \{P'\}\, sc\, \{Q'\}\end{array} \quad Q' \models Q}{\{P\}\, sc\, \{Q\}}\ \mathrm{conseq_{hoa}}$$

  Suppose $(\ell_0, zs_0, \sigma_0) \models P$ and $(\ell_0, zs_0, \sigma_0) \succ sc \rightarrow (\ell', zs', \sigma')$ for some $\ell_0, zs_0, \sigma_0, \ell', zs', \sigma'$ and α. As $P \models P'$, we get $(\ell_0, zs_0, \sigma_0) \models_\alpha P'$. By the induction hypothesis, therefore $(\ell', zs', \sigma') \models_\alpha Q'$. From $Q' \models Q$, this gives $(\ell', zs', \sigma') \models_\alpha Q$.

□

To get completeness, we have to assume that the underlying logical language is *expressive*. For any assertion Q, we need an assertion $\mathrm{wlp}(sc, Q)$ that, semantically, is its weakest precondition, i.e., for any state (ℓ, zs, σ) and valuation α of free variables, we have $(\ell, zs, \sigma) \models_\alpha \mathrm{wlp}(sc, Q)$ iff (i) $(\ell, zs, \sigma) \succ sc \rightarrow (\ell', zs', \sigma')$ implies $(\ell', zs', \sigma') \models_\alpha Q$ for any (ℓ', zs', σ') and (ii) there is no (ℓ', zs', σ') such that $(\ell, zs, \sigma) \succ sc \rightarrow\!\!\!\rightarrow (\ell', zs', \sigma')$. The wlp function is available for example when the underlying logical language has a greatest fixpoint operator.

Lemma 8 $\{\mathrm{wlp}(sc, Q)\}\, sc\, \{Q\}$.

Proof. We use structural induction on sc. There are the following cases.

- $sc = (\ell, \mathsf{load}\ x)$.

  By rule $\mathrm{load_{hoa}}$, we have

$$\{(pc = \ell \wedge Q[\ell + 1, x :: st/pc, st]) \vee (pc \neq \ell \wedge Q)\}\, (\ell, \mathsf{load}\ x)\, \{Q\}.$$

  We will show that we also have

$$\mathrm{wlp}((\ell, \mathsf{load}\ x), Q) \models (pc = \ell \wedge Q[\ell + 1, x :: st/pc, st]) \vee (pc \neq \ell \wedge Q)$$

  by showing that whenever $(\ell_0, zs_0, \sigma_0) \models_\alpha \mathrm{wlp}((\ell, \mathsf{load}\ x), Q)$, then also $(\ell_0, zs_0, \sigma_0) \models_\alpha (pc = \ell \wedge Q[\ell + 1, x :: st/pc, st]) \vee (pc \neq \ell \wedge Q)$. This will give us $\{\mathrm{wlp}((\ell, \mathsf{load}\ x), Q)\}\, (\ell, \mathsf{load}\ x)\, \{Q\}$ by using the rule of consequence (the argument will be similar for other instructions).

  Indeed, suppose $(\ell_0, zs_0, \sigma_0) \models_\alpha \mathrm{wlp}((\ell, \mathsf{load}\ x), Q)$ for some ℓ_0, zs_0, σ_0 and α. If $\ell_0 = \ell$, then by rule $\mathrm{load_{ns}}$ we have $(\ell_0, zs_0, \sigma_0) \succ (\ell, \mathsf{load}\ x) \rightarrow (\ell_0 + 1, \sigma_0(x) :: zs_0, \sigma_0)$, so $(\ell_0 + 1, \sigma_0(x) :: zs_0, \sigma_0) \models_\alpha Q$, from where $(\ell_0, zs_0, \sigma_0) \models_\alpha pc = \ell \wedge Q[\ell + 1, x :: st/pc, st]$. If $\ell_0 \neq \ell$, then by rule $\mathrm{ood_{ns}}$ we have $(\ell_0, zs_0, \sigma_0) \succ (\ell, \mathsf{load}\ x) \rightarrow (\ell_0, zs_0, \sigma_0)$, so $(\ell_0, zs_0, \sigma_0) \models_\alpha Q$, from where $(\ell_0, zs_0, \sigma_0) \models_\alpha pc \neq \ell \wedge Q$.

  By rule $\mathrm{conseq_{hoa}}$, we get

$$\{\mathrm{wlp}((\ell, \mathsf{load}\ x), Q)\}\, (\ell, \mathsf{load}\ x)\, \{Q\}$$

- $sc = (\ell, \mathsf{store}\ x)$: By rule store$_{\text{hoa}}$, we have

$$\left\{\begin{array}{ll} & (pc = \ell \wedge \exists z \in \mathbb{Z}, w \in (\mathbb{Z}+\mathbb{B})^*.\, st = z :: w \wedge Q[\ell+1, w, z/pc, st, x]) \\ \vee & (pc \neq \ell \wedge Q) \end{array}\right\} (\ell, \mathsf{store}\ x)\ \{\ Q\ \}$$

We also have $\text{wlp}((\ell, \mathsf{store}\ x), Q) \models pc = \ell \wedge \exists z \in \mathbb{Z}, w \in (\mathbb{Z}+\mathbb{B})^*.\, st = z :: w \wedge Q[\ell+1, w, z/pc, st, x]) \vee (pc \neq \ell \wedge Q)$. Indeed, suppose $(\ell_0, zs_0, \sigma_0) \models_\alpha \text{wlp}((\ell, \mathsf{store}\ x), Q)$ for some ℓ_0, zs_0, σ_0 and α. Then there must exist some zs'_0 such that $zs_0 = n :: zs'_0$ (since $(\ell_0, zs_0, \sigma_0) \models_\alpha \text{wlp}((\ell, \mathsf{store}\ x), Q)$ holds, it rules out abnormal termination from (ℓ_0, zs_0, σ_0) via $(\ell, \mathsf{store}\ x)$ by definition). If $\ell_0 = \ell$, then by rule store$_{\text{ns}}$ we have $(\ell_0, zs_0, \sigma_0) \succ\!(\ell, \mathsf{store}\ x)\!\rightarrow (\ell_0 + 1, zs'_0, \sigma_0[x \mapsto n])$, so $(\ell_0 + 1, zs'_0, \sigma_0[x \mapsto n]) \models_\alpha Q$, from where $(\ell_0, zs_0, \sigma_0) \models_\alpha pc = \ell \wedge \exists z \in \mathbb{Z}, w \in (\mathbb{Z}+\mathbb{B})^*.\, st = z :: w \wedge Q[\ell+1, w, z/pc, st, x])$. If $\ell_0 \neq \ell$, then by rule ood$_{\text{ns}}$ we have $(\ell_0, zs_0, \sigma_0) \succ\!(\ell, \mathsf{store}\ x)\!\rightarrow (\ell_0, zs_0, \sigma_0)$, so $(\ell_0, zs_0, \sigma_0) \models_\alpha pc \neq \ell \wedge Q$.

By rule conseq$_{\text{hoa}}$, we get

$$\{\text{wlp}((\ell, \mathsf{store}\ x), Q)\}\, (\ell, \mathsf{store}\ x)\, \{Q\}$$

- $sc = (\ell, \mathsf{push}\ n)$. Analogous to $\mathsf{load}\ x$ case.
- $sc = (\ell, \mathsf{add})$: By rule add$_{\text{hoa}}$, we have

$$\left\{\begin{array}{ll} & (pc = \ell \wedge \exists z_0, z_1 \in \mathbb{Z}, w \in (\mathbb{Z}+\mathbb{B})^*.\, st = z_0 :: z_1 :: w \\ & \wedge Q[\ell+1, z_0 + z_1 :: w/pc, st]) \\ \vee & (pc \neq \ell \wedge Q) \end{array}\right\} (\ell, \mathsf{add})\ \{\ Q\ \}$$

We also have $\text{wlp}((\ell, \mathsf{add}), Q) \models (pc = \ell \wedge \exists z_0, z_1 \in \mathbb{Z}, w \in (\mathbb{Z}+\mathbb{B})^*.\, st = z_0 :: z_1 :: w \wedge Q[\ell+1, z_0 + z_1 :: w/pc, st]) \vee (pc \neq \ell \wedge Q)$. Indeed, suppose $(\ell_0, zs_0, \sigma_0) \models_\alpha \text{wlp}((\ell, \mathsf{add}), Q)$ for some ℓ_0, zs_0, σ_0 and α. Then there must exist some zs'_0 such that $zs_0 = n_1 :: n_2 :: zs'_0$ (since $(\ell_0, zs_0, \sigma_0) \models_\alpha \text{wlp}((\ell, \mathsf{add}), Q)$ rules out abnormal termination from (ℓ_0, zs_0, σ_0) by definition). If $\ell_0 = \ell$, then by rule add$_{\text{ns}}$ we have $(\ell_0, zs_0, \sigma_0) \succ\!(\ell, \mathsf{add})\!\rightarrow (\ell_0 + 1, n_1 + n_2 :: zs'_0, \sigma_0)$, so $(\ell + 1, zs'_0, \sigma_0) \models_\alpha Q$, from where $(\ell_0, zs_0, \sigma_0) \models_\alpha pc = \ell \wedge \exists z_0, z_1 \in \mathbb{Z}, w \in (\mathbb{Z}+\mathbb{B})^*.\, st = z_0 :: z_1 :: w \wedge Q[\ell+1, z_0 + z_1 :: w/pc, st]$. If $\ell_0 \neq \ell$, then by rule ood$_{\text{ns}}$ we have $(\ell_0, zs_0, \sigma_0) \succ\!(\ell, \mathsf{add})\!\rightarrow (\ell_0, zs_0, \sigma_0)$, so $(\ell_0, zs_0, \sigma_0) \models_\alpha pc \neq \ell \wedge Q$.

By rule conseq$_{\text{hoa}}$, we get

$$\{\text{wlp}((\ell, \mathsf{add}), Q)\}\, (\ell, \mathsf{add})\, \{Q\}$$

- $sc = (\ell, \mathsf{goto}\ m)$: By rule goto$_{\text{hoa}}$, we have

$$\left\{\begin{array}{ll} & (pc = \ell \wedge ((m \neq \ell \wedge Q[m/pc]) \vee m = \ell)) \\ \vee & (pc \neq \ell \wedge Q) \end{array}\right\} (\ell, \mathsf{goto}\ m)\ \{\ Q\ \}$$

We also have

$$\text{wlp}((\ell, \mathsf{goto}\ m), Q) \models (pc = \ell \wedge (Q[m/pc] \vee m = \ell)) \vee (pc \neq \ell \wedge Q).$$

Indeed, suppose $(\ell_0, zs_0, \sigma_0) \models_\alpha \mathrm{wlp}((\ell, \mathsf{goto}\ m), Q)$ for some ℓ_0, zs_0, σ_0 and α. If $\ell_0 = \ell$ and $m \neq \ell$, then by rule $\mathrm{goto}_{\mathrm{ns}}$ we have $(\ell_0, zs_0, \sigma_0) \succ (\ell, \mathsf{goto}\ m) \to (m, zs_0, \sigma_0)$, so $(m, zs_0, \sigma_0) \models_\alpha Q$, from where $(\ell_0, zs_0, \sigma_0) \models_\alpha pc = \ell \wedge Q[m/pc]$. If $\ell_0 = \ell$ and $m = \ell$, then $(\ell_0, zs_0, \sigma_0) \models_\alpha pc = \ell \wedge m = \ell$. If $\ell_0 \neq \ell$, then by rule $\mathrm{ood}_{\mathrm{ns}}$ we have $(\ell_0, zs_0, \sigma_0) \succ (\ell, \mathsf{goto}\ m) \to (\ell_0, zs_0, \sigma_0)$, so $(\ell_0, zs_0, \sigma_0) \models_\alpha pc \neq \ell \wedge Q$.

By rule $\mathrm{conseq}_{\mathrm{hoa}}$, we get

$$\{\mathrm{wlp}((\ell, \mathsf{goto}\ m), Q)\}\ (\ell, \mathsf{goto}\ m)\ \{Q\}$$

- $sc = (\ell, \mathsf{gotoF}\ m)$: By rule $\mathrm{gotoF}_{\mathrm{hoa}}$, we have

$$\dfrac{}{\left\{\begin{array}{llll} & (pc = \ell & \wedge((m \neq \ell & \wedge((\exists w \in (\mathbb{Z}+\mathbb{B})^*.\ st = \mathrm{tt} :: w \\ & & & \quad \wedge Q[\ell+1, w/pc, st]) \\ & & & \vee(\exists w \in (\mathbb{Z}+\mathbb{B})^*.\ st = \mathrm{ff} :: w \\ & & & \quad \wedge Q[m, w/pc, st]))) \\ & & \vee(m = \ell & \wedge \exists \mathit{ffs} \in \{\mathrm{ff}\}^*, w \in (\mathbb{Z}+\mathbb{B})^*. st = \mathit{ffs} +\!\!+\ \mathrm{tt} :: w \\ & & & \quad \wedge Q[\ell+1, w/pc, st]))) \\ \vee & (pc \neq \ell & \wedge Q) & \end{array}\right\} (\ell, \mathsf{gotoF}\ m)\ \{\ Q\ \}}$$

We also have

$$\begin{array}{llll} \multicolumn{4}{l}{\mathrm{wlp}((\ell, \mathsf{gotoF}\ m), Q) \models} \\ & (pc = \ell & \wedge((m \neq \ell & \wedge((\exists w \in (\mathbb{Z}+\mathbb{B})^*.\ st = \mathrm{tt} :: w \\ & & & \quad \wedge Q[\ell+1, w/pc, st]) \\ & & & \vee(\exists w \in (\mathbb{Z}+\mathbb{B})^*.\ st = \mathrm{ff} :: w \\ & & & \quad \wedge Q[m, w/pc, st]))) \\ & & \vee(m = \ell & \wedge \exists \mathit{ffs} \in \{\mathrm{ff}\}^*, w \in (\mathbb{Z}+\mathbb{B})^*. st = \mathit{ffs} +\!\!+\ \mathrm{tt} :: w \\ & & & \quad \wedge Q[\ell+1, w/pc, st]))) \\ \vee & (pc \neq \ell & \wedge Q) & \end{array}$$

Indeed, suppose $(\ell_0, zs_0, \sigma_0) \models_\alpha \mathrm{wlp}((\ell, \mathsf{gotoF}\ m), Q)$ for some ℓ_0, zs_0, σ_0 and α. If $\ell_0 = \ell$ and $m \neq \ell$ and zs_0 is of the form $\mathrm{tt} :: zs_0'$, then by rule $\mathrm{gotoF}_{\mathrm{ns}}^{\neq\mathrm{tt}}$ we have $(\ell_0, zs_0, \sigma_0) \succ (\ell, \mathsf{gotoF}\ m) \to (\ell+1, zs_0', \sigma_0)$, so $(\ell+1, zs_0', \sigma_0) \models_\alpha Q$, from where $(\ell_0, zs_0, \sigma_0) \models_\alpha pc = \ell \wedge ((m \neq \ell \wedge ((\exists w \in (\mathbb{Z}+\mathbb{B})^*.\ st = \mathrm{tt} :: w \wedge Q[\ell+1, w/pc, st])))))$.

If $\ell_0 = \ell$ and $m \neq \ell$ and zs_0 is of the form $\mathrm{ff} :: zs_0'$, then by rule $\mathrm{gotoF}_{\mathrm{ns}}^{\neq\mathrm{ff}}$ we have $(\ell_0, zs_0, \sigma_0) \succ (\ell, \mathsf{gotoF}\ m) \to (m, zs_0', \sigma_0)$, so $(m, zs_0', \sigma_0) \models_\alpha Q$, from where again $(\ell_0, zs_0, \sigma_0) \models_\alpha pc = \ell \wedge ((m \neq \ell \wedge ((\exists w \in (\mathbb{Z}+\mathbb{B})^*.\ st = \mathrm{ff} :: w \wedge Q[m, w/pc, st]))$.

If $\ell_0 = \ell$ and $m = \ell$ and zs_0 is of the form $\mathrm{ff} :: ... :: \mathrm{ff} :: \mathrm{tt} :: zs_0'$ then by rule $\mathrm{gotoF}_{\mathrm{ns}}^{=}$ we have $(\ell_0, zs_0, \sigma_0) \succ (\ell, \mathsf{gotoF}\ m) \to (\ell+1, zs_0', \sigma_0)$, so $(\ell+1, zs_0', \sigma_0) \models_\alpha Q$, from where $(\ell_0, zs_0, \sigma_0) \models_\alpha pc = \ell \wedge (m = \ell \wedge \exists \mathit{ffs} \in \{\mathrm{ff}\}^*, w \in (\mathbb{Z}+\mathbb{B})^*. st = \mathit{ffs} +\!\!+\ \mathrm{tt} :: w \wedge Q[\ell+1, w/pc, st])$.

We know that zs_0 cannot be of any other form, since the definition of wlp rules out abnormal termination.

If $\ell_0 \neq \ell$, then by rule $\mathrm{ood}_{\mathrm{ns}}$ we have $(\ell_0, zs_0, \sigma_0) \succ\!(\ell, \mathsf{gotoF}\ m)\!\rightarrow (\ell_0, zs_0, \sigma_0)$, so $(\ell_0, zs_0, \sigma_0) \models_\alpha pc \neq \ell \wedge Q$.

By rule $\mathrm{conseq}_{\mathrm{hoa}}$, we get

$$\{\mathrm{wlp}((\ell, \mathsf{gotoF}\ m), Q)\}\,(\ell, \mathsf{gotoF}\ m)\,\{Q\}$$

- $sc = \mathbf{0}$: By rule $\mathbf{0}_{\mathrm{hoa}}$, we have

  $$\{\mathrm{wlp}(\mathbf{0}, Q)\}\,\mathbf{0}\,\{\mathrm{wlp}(\mathbf{0}, Q)\}$$

  We also have

  $$\mathrm{wlp}(\mathbf{0}, Q) \models Q$$

  Indeed, suppose $(\ell_0, zs_0, \sigma_0) \models_\alpha \mathrm{wlp}(\mathbf{0}, Q)$ for some ℓ_0, zs_0, σ_0 and α. Rule $\mathrm{ood}_{\mathrm{ns}}$ gives us $(\ell_0, zs_0, \sigma_0) \succ\!\mathbf{0}\!\rightarrow (\ell_0, zs_0, \sigma_0)$ and hence $(\ell_0, zs_0, \sigma_0) \models_\alpha Q$.

  Hence by rule $\mathrm{conseq}_{\mathrm{hoa}}$, we get

  $$\{\mathrm{wlp}(\mathbf{0}, Q)\}\,\mathbf{0}\,\{Q\}$$

- $sc = sc_0 \oplus sc_1$: By the induction hypothesis, we have

  $$\{\mathrm{wlp}(sc_i, \mathrm{wlp}(sc_0 \oplus sc_1, Q))\}\,sc_i\,\{\mathrm{wlp}(sc_0 \oplus sc_1, Q)\}$$

  (for $i = 0$ and 1). We also have

  $$pc \in \mathrm{dom}(sc_i) \wedge \mathrm{wlp}(sc_0 \oplus sc_1, Q) \models \mathrm{wlp}(sc_i, \mathrm{wlp}(sc_0 \oplus sc_1, Q))$$

  (for $i = 0$ and 1). Indeed, suppose $(\ell_0, zs_0, \sigma_0) \models_\alpha pc \in \mathrm{dom}(sc_i) \wedge \mathrm{wlp}(sc_0 \oplus sc_1, Q)$ for some ℓ_0, zs_0, σ_0 and α. Then $\ell_0 \in \mathrm{dom}(sc_i)$ and $(\ell_0, zs_0, \sigma_0) \models_\alpha \mathrm{wlp}(sc_0 \oplus sc_1, Q)$. Consider any ℓ'', zs'', σ'', ℓ', zs', σ' such that $(\ell_0, zs_0, \sigma_0) \succ\!sc_i\!\rightarrow (\ell'', zs'', \sigma'')$ and $(\ell'', zs'', \sigma'') \succ\!sc_0 \oplus sc_1\!\rightarrow (\ell', zs', \sigma')$. By rule $\oplus^i_{\mathrm{ns}}$ we have $(\ell_0, zs_0, \sigma_0) \succ\!sc_0 \oplus sc_1\!\rightarrow (\ell', zs', \sigma')$, which gives us $(\ell', zs', \sigma') \models_\alpha Q$. We can also see that there can be no abnormal termination for sc_i from state (ℓ_0, zs_0, σ_0), since then by rule $\oplus^{abn}_{\mathrm{ns}}$, we would get abnormal termination for $sc_0 \oplus sc_1$, which is impossible due to $(\ell_0, zs_0, \sigma_0) \models_\alpha \mathrm{wlp}(sc_0 \oplus sc_1, Q)$. Furthermore, abnormal termination from (ℓ'', zs'', σ'') by $sc_0 \oplus sc_1$ is also impossible, since that would yield abnormal termination for $sc_0 \oplus sc_1$ from state (ℓ_0, zs_0, σ_0) via $\oplus^{abl}_{\mathrm{ns}}$ rule, which is impossible due to $(\ell_0, zs_0, \sigma_0) \models_\alpha \mathrm{wlp}(sc_0 \oplus sc_1, Q)$. Hence, $(\ell_0, zs_0, \sigma_0) \models_\alpha \mathrm{wlp}(sc_i, \mathrm{wlp}(sc_0 \oplus sc_1, Q))$ as needed.

  Rule $\mathrm{conseq}_{\mathrm{hoa}}$ gives us

  $$\{pc \in \mathrm{dom}(sc_i) \wedge \mathrm{wlp}(sc_0 \oplus sc_1, Q)\}\,sc_i\,\{\mathrm{wlp}(sc_0 \oplus sc_1, Q)\}$$

  (for $i = 0$ and 1). From here, rule $\oplus_{\mathrm{hoa}}$ gives us

  $$\{\mathrm{wlp}(sc_0 \oplus sc_1, Q)\}\,sc_0 \oplus sc_1\,\left\{\begin{array}{l} \quad pc \notin \mathrm{dom}(sc_0) \wedge pc \notin \mathrm{dom}(sc_1) \\ \wedge \ \ \mathrm{wlp}(sc_0 \oplus sc_1, Q) \end{array}\right\}$$

Further, we also have

$$pc \notin \text{dom}(sc_0) \wedge pc \notin \text{dom}(sc_1) \wedge \text{wlp}(sc_0 \oplus sc_1, Q) \models Q$$

Indeed, suppose $(\ell_0, zs_0, \sigma_0) \models_\alpha pc \notin \text{dom}(sc_0) \wedge pc \notin \text{dom}(sc_1) \wedge \text{wlp}(sc_0 \oplus sc_1, Q)$ for some ℓ_0, zs_0, σ_0 and α. We then have $\ell_0 \notin \text{dom}(sc_0 \oplus sc_1)$ and $(\ell_0, zs_0, \sigma_0) \models_\alpha \text{wlp}(sc_0 \oplus sc_1, Q)$. Rule ood_{ns} gives us $(\ell_0, zs_0, \sigma_0) \succ sc_0 \oplus sc_1 \rightarrow (\ell_0, zs_0, \sigma_0)$ and hence $(\ell_0, zs_0, \sigma_0) \models_\alpha Q$.

Hence by rule $\text{conseq}_{\text{hoa}}$, we get

$$\{\text{wlp}(sc_0 \oplus sc_1, Q)\}\, sc_0 \oplus sc_1\, \{Q\}$$

□

Theorem 8 (Completeness of Hoare logic) *If, for any (ℓ, zs, σ) and α such that $(\ell, zs, \sigma) \models_\alpha P$, it holds that (i) for any (ℓ', zs', σ') such that $(\ell, zs, \sigma) \succ sc \rightarrow (\ell', zs', \sigma')$, we have $(\ell', zs', \sigma') \models_\alpha Q$, and (ii) there is no (ℓ', zs', σ') such that $(\ell, zs, \sigma) \succ sc \rightarrowtail (\ell', zs', \sigma')$, then $\{P\}\, sc\, \{Q\}$.*

Proof. Immediate from the lemma using that any precondition of a code wrt. a postcondition entails its wlp. □

While we have introduced the semantics and logic for a language with fixed jump targets, our approach applies equally well to a version where indirect jumps (also called embedded code pointers) are allowed. The modifications required are trivial. In the instruction syntax, the jump targets would not be given as constant labels in instructions, but would be popped off the stack. Thus the instruction $\mathsf{goto}\ m$ would become goto, which branches to the label given on top of the stack. The natural semantics rule for this instruction would be

$$\frac{m \in \mathbb{Z}}{(\ell, m :: zs, \sigma) \succ (\ell, \mathsf{goto}) \rightarrow (m, zs, \sigma)}\ \text{goto}_{\text{ns}}$$

It would also need an abnormal termination rule, since an infinite loop via the goto instructions (by a jump back on itself) would not be possible anymore: the stack would sooner or later run empty, giving abnormal termination.

For conditional jumps the rule change would be similar, so the jump target would be popped off the stack in addition to the jump guard.

The Hoare rule for goto would become

$$\frac{}{\left\{ \begin{array}{ll} & (pc = \ell \wedge \exists m \in \mathbb{Z}, w \in (\mathbb{Z} + \mathbb{B})^*, ls \in \{\ell\}^*.\ st = ls +\!\!+\, m :: w \wedge m \neq \ell \\ & \quad \wedge Q[m, w/pc, st]) \\ \vee & (pc \neq \ell \wedge Q) \end{array} \right\} (\ell, \mathsf{goto})\ \{\ Q\ \}}\ \text{goto}_{\text{hoa}}$$

This is somewhat similar to Java bytecode `jsr`/`ret` instructions: `jsr` branches to a given label in the method code and pushes a return address to the following instruction, while `ret` recovers a return address from a register and branches to the corresponding instruction. So jsr would be equivalent to the pair of instruction $(\ell, \mathsf{push}\ \ell{+}2), (\ell{+}1, \mathsf{goto}\ m)$, and ret would be equivalent to $(\ell, \mathsf{load}\ retAddress), (\ell{+}1, \mathsf{goto})$.

3.3 A dip into type systems

The logic we presented is error-free: a Hoare triple guarantees, in addition to the program satisfying the pre- and post condition relations, that it cannot have a runtime error. Naturally, this logic can be weakened into a type system, which *only* guarantees that type errors cannot occur. In this section, we describe a compositional type system guaranteeing error-freedom for SPush programs. We will not delve deeply into the meta-theoretic properties of the type system, rather, this section serves to demonstrate how the compositional approach is applicable for reasoning about low-level code in general, not just for presenting the logic.

Instead of relating assertions as Hoare triples do, typings relate state types. The intuitive meaning of a typing is analogous to that of a Hoare triple: it says that if the given piece of code is run from an initial state in the given pretype, then if it terminates normally, the final state is in the posttype, and, moreover, it cannot terminate abnormally. Contrarily to assertions, state types are designed to record only that state information that is necessary for guaranteeing error-freedom.

The building blocks for state types are value types $\tau \in \mathbf{ValType}$ and stack types $\Psi \in \mathbf{StackType}$, defined by the grammars

$$\begin{array}{rcl} \tau & ::= & \bot \mid \mathrm{int} \mid \mathrm{bool} \mid ? \\ \Psi & ::= & \bot \mid [] \mid \tau :: \Psi \mid * \end{array}$$

(note the overloading of the $\bot$ sign). A state type $\Pi \in \mathbf{StateType}$ is a finite set of labelled stack types, i.e., pairs of a label and a stack type: $\mathbf{StateType} =_{\mathrm{df}} \mathcal{P}_{\mathrm{fin}}(\mathbf{Label} \times \mathbf{StackType})$. A state type Π is wellformed iff no label ℓ in it labels more than one stack type, i.e., $(\ell, \Psi) \in \Pi$ and $(\ell, \Psi') \in \Pi$ imply $\Psi = \Psi'$. The domain $\mathrm{dom}(\Pi)$ of a state type is the set of labels appearing in it, i.e., $\mathrm{dom}(\Pi) =_{\mathrm{df}} \{\ell \mid (\ell, \Psi) \in \Pi\}$.

We will use the notation $\Pi\lceil_L$ for the restriction of a state type Π to a domain $L \subseteq \mathbf{Label}$, i.e., $\Pi\lceil_L =_{\mathrm{df}} \{(\ell, \Psi) \mid (\ell, \Psi) \in \Pi, \ell \in L\}$, and write $\overline{L}$ for the complement of L, i.e., $\overline{L} =_{\mathrm{df}} \mathbf{Label} \backslash L$.

The meanings of value, stack and state types are set-theoretic, they denote sets of abstract values, abstract stacks and abstract states. The semantic functions $(\!| - |\!) \in \mathbf{ValType} \to \mathcal{P}(\{\mathrm{int}, \mathrm{bool}\})$, $(\!| - |\!) \in \mathbf{StackType} \to \mathcal{P}(\mathbf{AbsStack})$, $(\!| - |\!) \in \mathbf{StateType} \to \mathcal{P}(\mathbf{AbsState})$ are defined as follows:

$$\frac{}{\tau \leq \tau} \quad \frac{}{\bot \leq \tau} \quad \frac{}{\tau \leq ?}$$

$$\frac{}{\Psi \leq \Psi} \quad \frac{\Psi \leq \Psi'' \quad \Psi'' \leq \Psi'}{\Psi \leq \Psi'} \quad \frac{}{\bot :: \Psi \leq \bot} \quad \frac{}{\tau :: \bot \leq \bot} \quad \frac{}{\bot \leq \Psi} \quad \frac{}{\Psi \leq *} \quad \frac{\tau \leq \tau' \quad \Psi \leq \Psi'}{\tau :: \Psi \leq \tau' :: \Psi'}$$

$$\frac{\forall \ell, \Psi.\, (\ell, \Psi) \in \Pi \Rightarrow \Psi \leq \bot \vee \exists \Psi'.\, (\ell, \Psi') \in \Pi' \wedge \Psi \leq \Psi'}{\Pi \leq \Pi'}$$

Figure 3.3: Subtyping rules of SPush

$$\begin{aligned}
(\!|\, \bot \,|\!) &=_{\mathrm{df}} \emptyset \\
(\!|\, \mathrm{int} \,|\!) &=_{\mathrm{df}} \{\mathrm{int}\} \\
(\!|\, \mathrm{bool} \,|\!) &=_{\mathrm{df}} \{\mathrm{bool}\} \\
(\!|\, ? \,|\!) &=_{\mathrm{df}} \{\mathrm{int}, \mathrm{bool}\} \\
(\!|\, \bot \,|\!) &=_{\mathrm{df}} \emptyset \\
(\!|\, [] \,|\!) &=_{\mathrm{df}} \{[]\} \\
(\!|\, \tau :: \Psi \,|\!) &=_{\mathrm{df}} \{\delta :: \psi \mid \delta \in (\!|\, \tau \,|\!), \psi \in (\!|\, \Psi \,|\!)\} \\
(\!|\, * \,|\!) &=_{\mathrm{df}} \{\mathrm{int}, \mathrm{bool}\}^* \\
(\!|\, \Pi \,|\!) &=_{\mathrm{df}} \{(\ell, \psi) \mid (\ell, \Psi) \in \Pi, \psi \in (\!|\, \Psi \,|\!)\}
\end{aligned}$$

On each of the three categories of types, we define a subtyping relation by the appropriate rules in Figure 3.3. These are relations $\leq \subseteq \mathbf{ValType} \times \mathbf{ValType}$, $\leq \subseteq \mathbf{StackType} \times \mathbf{StackType}$, $\leq \subseteq \mathbf{StateType} \times \mathbf{StateType}$.

The typing relation $- : \longrightarrow \subseteq \mathbf{StateType} \times \mathbf{SCode} \times \mathbf{StateType}$ is defined by the rules in Figure 3.4. The typing rules for instructions are presented in a "weakest pretype" style, where the pretype is obtained by applying appropriate substitutions in the given posttype. For example the rule $\mathrm{load}_{\mathrm{ts}}$ for $(\ell, \mathsf{load}\ x)$ states that if stack type $\tau :: \Psi$ (where τ is int or ?) or $*$ is required at label $\ell + 1$, then the suitable stack types for label ℓ are Ψ and $*$, respectively. Any other posttype at label $\ell + 1$ does not have a suitable pretype. At first sight, it might seem that wellformedness can be lost in the pretype by taking the union. This is in fact not the case: there is at most one stack type associated with label $\ell + 1$ in Π, hence both sets have at most one element and one of them must be empty. The rest of the non-jump instruction rules are defined in similar fashion.

The jump rules might need some explanation. The $\mathrm{goto}^{=}_{\mathrm{ts}}$ rule allows to derive pretype $*$ for label ℓ: since the instruction does not terminate, any posttype will be satisfied by any pretype at label ℓ. The $\mathrm{gotoF}^{\neq}_{\mathrm{ts}}$ rule combines two posttypes; since gotoF can branch, both posttypes must be satisfied at the entry, meaning that the pretype is the intersection of the posttypes. No pretype at ℓ can guarantee any posttype in the case of $(\ell, \mathsf{gotoF}\ \ell)$, since such instruction could always terminate abnormally. The consequence rule could also be called subsumption, given that we are speaking about a type system: that is what it is really.

$$\frac{}{(\ell, \mathsf{load}\ x) : \{(\ell, \Psi) \mid (\ell+1, \tau :: \Psi) \in \Pi, \mathrm{int} \leq \tau\} \cup \{(\ell, *) \mid (\ell+1, *) \in \Pi\} \cup \Pi\restriction_{\overline{\{\ell\}}} \longrightarrow \Pi} \ \mathrm{load_{ts}}$$

$$\frac{}{(\ell, \mathsf{store}\ x) : \{(\ell, \mathrm{int} :: \Psi) \mid (\ell+1, \Psi) \in \Pi\} \cup \Pi\restriction_{\overline{\{\ell\}}} \longrightarrow \Pi} \ \mathrm{store_{ts}}$$

$$\frac{}{(\ell, \mathsf{push}\ n) : \{(\ell, \Psi) \mid (\ell+1, \tau :: \Psi) \in \Pi, \mathrm{int} \leq \tau\} \cup \{(\ell, *) \mid (\ell+1, *) \in \Pi\} \cup \Pi\restriction_{\overline{\{\ell\}}} \longrightarrow \Pi} \ \mathrm{push_{ts}}$$

$$\frac{}{(\ell, \mathsf{add}) : \begin{array}{ll} & \{(\ell, \mathrm{int} :: \mathrm{int} :: \Psi) \mid (\ell+1, \tau :: \Psi) \in \Pi, \mathrm{int} \leq \tau\} \\ \cup & \{(\ell, \mathrm{int} :: \mathrm{int} :: *) \mid (\ell+1, *) \in \Pi\} \end{array} \cup \Pi\restriction_{\overline{\{\ell\}}} \longrightarrow \Pi} \ \mathrm{add_{ts}}$$

$$\cdots$$

$$\frac{m \neq \ell}{(\ell, \mathsf{goto}\ m) : \{(\ell, \Psi) \mid (m, \Psi) \in \Pi\} \cup \Pi\restriction_{\overline{\{\ell\}}} \longrightarrow \Pi} \ \mathrm{goto_{ts}^{\neq}}$$

$$\frac{}{(\ell, \mathsf{goto}\ \ell) : \{(\ell, *)\} \cup \Pi\restriction_{\overline{\{\ell\}}} \longrightarrow \Pi} \ \mathrm{goto_{ts}^{=}}$$

$$\frac{m \neq \ell}{(\ell, \mathsf{gotoF}\ m) : \{(\ell, \mathrm{bool} :: (\Psi \wedge \Psi')) \mid (\ell+1, \Psi), (m, \Psi') \in \Pi\} \cup \Pi\restriction_{\overline{\{\ell\}}} \longrightarrow \Pi} \ \mathrm{gotoF_{ts}^{\neq}}$$

$$\frac{}{(\ell, \mathsf{gotoF}\ \ell) : \Pi\restriction_{\overline{\{\ell\}}} \longrightarrow \Pi} \ \mathrm{gotoF_{ts}^{=}}$$

$$\frac{}{\mathbf{0} : \Pi \longrightarrow \Pi} \ \mathbf{0}_{\mathrm{ts}} \qquad \frac{sc_0 : \Pi\restriction_{\mathrm{dom}(sc_0)} \longrightarrow \Pi \quad sc_1 : \Pi\restriction_{\mathrm{dom}(sc_1)} \longrightarrow \Pi}{sc_0 \oplus sc_1 : \Pi \longrightarrow \Pi\restriction_{\overline{\mathrm{dom}(sc_0) \cup \mathrm{dom}(sc_1)}}} \ \oplus_{\mathrm{ts}}$$

$$\frac{\Pi_0' \leq \Pi_0 \quad sc : \Pi_0 \longrightarrow \Pi_1 \quad \Pi_1 \leq \Pi_1'}{sc : \Pi_0' \longrightarrow \Pi_1'} \ \mathrm{conseq_{ts}}$$

Figure 3.4: Typing rules of SPush

The type system is sound wrt. the natural semantics in the sense of error-free partial correctness.

Let us a have a function abs that relates a concrete state to an abstract state: $\mathrm{abs} \in \mathbf{State} \to \mathbf{AbsState}$, defined by $\mathrm{abs}(\ell, zs, \sigma) =_{\mathrm{df}} (\ell, \mathrm{abs}(zs))$ where $\mathrm{abs} \in \mathbf{Stack} \to \mathbf{AbsStack}$ replaces concrete values in a stack with the names of their types: $\mathrm{abs}([]) =_{\mathrm{df}} []$, $\mathrm{abs}(n :: zs) =_{\mathrm{df}} \mathrm{int} :: \mathrm{abs}(zs)$ for $n \in \mathbb{Z}$, and $\mathrm{abs}(b :: zs) =_{\mathrm{df}} \mathrm{bool} :: \mathrm{abs}(zs)$ for $b \in \mathbb{B}$.

Theorem 9 (Soundness of typing) *If $sc : \Pi \longrightarrow \Pi'$ and $\mathrm{abs}(\ell, zs, \sigma) \in (\!|\, \Pi \,|\!)$, then (i) for any (ℓ', zs', σ') such that $(\ell, zs, \sigma) \succ sc \rightarrow (\ell', zs', \sigma')$, we have $\mathrm{abs}(\ell', zs', \sigma') \in (\!|\, \Pi' \,|\!)$, and (ii) there is no (ℓ', zs', σ') such that $(\ell, zs, \sigma) \succ sc \rightarrow\!\!\!\!\mid\ (\ell', zs', \sigma')$.*

It would also possible to show that the type system is complete wrt. the abstract natural semantics, which operates on type names as abstract values instead of concrete types.

It is fairly obvious that state types can be translated to assertions. We can define concretization functions $\mathrm{conc} \in \mathbf{ValType} \to \mathcal{P}(\mathbb{Z}+\mathbb{B})$, $\mathrm{conc} \in \mathbf{StackType} \to \mathcal{P}(\mathbf{Stack})$, $\mathrm{conc} \in \mathbf{StateType} \to \mathbf{Assn}$, taking us from the language of the type

system to the language of the logic, by

$$\begin{array}{rcl}
\text{conc}(\bot) & =_{\text{df}} & \emptyset \\
\text{conc}(\text{int}) & =_{\text{df}} & \mathbb{Z} \\
\text{conc}(\text{bool}) & =_{\text{df}} & \mathbb{B} \\
\text{conc}(?) & =_{\text{df}} & \mathbb{Z} + \mathbb{B} \\
\text{conc}(\bot) & =_{\text{df}} & \emptyset \\
\text{conc}([]) & =_{\text{df}} & \{[]\} \\
\text{conc}(\tau :: \Psi) & =_{\text{df}} & \{z :: w \mid z \in \text{conc}(\tau), w \in \text{conc}(\Psi)\} \\
\text{conc}(*) & =_{\text{df}} & (\mathbb{Z} + \mathbb{B})^* \\
\text{conc}(\Pi) & =_{\text{df}} & \\
\multicolumn{3}{c}{\bigvee\{pc = \ell \wedge st \in \text{conc}(\Psi) \mid (\ell, \Psi) \in \Pi\}}
\end{array}$$

Concretization preserves and reflects derivable subtypings/entailments.

Theorem 10 (Preservation of subtypings and reflection of entailments by concretization)
(i) $\tau \leq \tau'$ **iff** $\text{conc}(\tau) \models \text{conc}(\tau')$**.** **(ii)** $\Psi \leq \Psi'$ **iff** $\text{conc}(\Psi) \models \text{conc}(\Psi')$**.** **(iii)** $\Pi \leq \Pi'$ **iff** $\text{conc}(\Pi) \models \text{conc}(\Pi')$**.**

Preservation holds also of typing.

Theorem 11 (Preservation of typings by concretization) *If* $sc : \Pi \longrightarrow \Pi'$, *then* $\{\text{conc}(\Pi)\}\, sc\, \{\text{conc}(\Pi')\}$.

We do not get reflection of Hoare triples by concretization, however. Consider, for example, the code $sc =_{\text{df}} (0, \mathsf{push}\ \text{tt}) \oplus ((1, \mathsf{gotoF}\ 3) \oplus (2, \mathsf{push}\ 17))$. We have $\text{conc}((0, [])) = pc = 0 \wedge st = []$, $\text{conc}((3, [\text{int}])) = pc = 3 \wedge \exists z \in \mathbb{Z}.\ st = [z]$ and can derive $\{pc = 0 \wedge st = []\}\, sc\, \{pc = 3 \wedge \exists z \in \mathbb{Z}.\ st = [z]\}$, while we cannot derive $sc : \{(0, [])\} \longrightarrow \{(3, [\text{int}])\}$. The type system does not discover that the false branch will never be taken. The best posttype we can get for $\{(0, [])\}$ is $\{(3, *)\}$.

3.4 Compilation

This section is a first look into proof transformation. We shall define a compilation function from While programs to SPush pieces of code. and show that it preserves and reflects evaluations. Moreover, we shall also show that compilation preserves and reflects derivable Hoare triples. Non-constructively this is obvious because the logics of both While and SPush are sound and complete. We show that compilation also preserves and reflects the actual Hoare triple derivations that establish derivability, thus effectively allowing for compilation of proofs.

The compilation function is standard except that it produces structured code (we have chosen structures that are the most convenient for us) and is compositional. The compilation rules are given in Figure 3.5. The compilation relation for expressions $- \searrow - \subseteq \textbf{Label} \times (\textbf{AExp} \cup \textbf{BExp}) \times \textbf{SCode} \times \textbf{Label}$ relates a label and a

$$\frac{}{n\ {}^{\ell}\!\searrow_{\ell+1} (\ell, \mathsf{push}\ n)} \quad \frac{}{x\ {}^{\ell}\!\searrow_{\ell+1} (\ell, \mathsf{load}\ x)} \quad \frac{a_0\ {}^{\ell}\!\searrow_{\ell''} sc_0 \quad a_1\ {}^{\ell''}\!\searrow_{\ell'} sc_1}{a_0 + a_1\ {}^{\ell}\!\searrow_{\ell'+1} (sc_0 \oplus sc_1) \oplus (\ell', \mathsf{add})}$$

$$\frac{b_0\ {}^{\ell}\!\searrow_{\ell''} sc_0 \quad b_1\ {}^{\ell''}\!\searrow_{\ell'} sc_1}{b_0 = b_1\ {}^{\ell}\!\searrow_{\ell'+1} (sc_0 \oplus sc_1) \oplus (\ell', \mathsf{eq})}$$

$$\frac{a\ {}^{\ell}\!\searrow_{\ell'} sc}{x := a\ {}^{\ell}\!\searrow_{\ell'+1} (sc \oplus (\ell', \mathsf{store}\ x))} \quad \frac{}{\mathsf{skip}\ {}^{\ell}\!\searrow_{\ell} \mathbf{0}} \quad \frac{s_0\ {}^{\ell}\!\searrow_{\ell''} sc_0 \quad s_1\ {}^{\ell''}\!\searrow_{\ell'} sc_1}{s_0; s_1\ {}^{\ell}\!\searrow_{\ell'} sc_0 \oplus sc_1}$$

$$\frac{b\ {}^{\ell}\!\searrow_{\ell''} sc_b \quad s_t\ {}^{\ell''+1}\!\searrow_{\ell'''} sc_t \quad s_f\ {}^{\ell'''+1}\!\searrow_{\ell'} sc_f}{\mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f\ {}^{\ell}\!\searrow_{\ell'} (sc_b \oplus (\ell'', \mathsf{gotoF}\ \ell'''+1)) \oplus ((sc_t \oplus (\ell''', \mathsf{goto}\ \ell')) \oplus sc_f)}$$

$$\frac{b\ {}^{\ell}\!\searrow_{\ell''} sc_b \quad s\ {}^{\ell''+1}\!\searrow_{\ell'} sc}{\mathsf{while}\ b\ \mathsf{do}\ s\ {}^{\ell}\!\searrow_{\ell'+1} (sc_b \oplus (\ell'', \mathsf{gotoF}\ \ell'+1)) \oplus (sc \oplus (\ell', \mathsf{goto}\ \ell))}$$

Figure 3.5: Rules of compilation from While to SPush

While expression to a piece of code and another label. The relation for statements $- \searrow - \subseteq \mathbf{Label} \times \mathbf{Stm} \times \mathbf{SCode} \times \mathbf{Label}$ is similar. The idea is that the domain of a compiled expression or statement will be a left-closed, right-open interval. (It may be an empty interval, which does not even contain its beginning-point.) The first label is the beginning-point of the interval and the second is the corresponding end-point.

Compilation is total and deterministic and produces a piece of code whose support is exactly the desired interval.

Lemma 9 (Totality and determinacy of compilation)
(i) For any ℓ, a, there exist sc, ℓ' such that $a\ {}^{\ell}\!\searrow_{\ell'} sc$. If $a\ {}^{\ell}\!\searrow_{\ell_0} sc_0$ and $a\ {}^{\ell}\!\searrow_{\ell_1} sc_1$, then $sc_0 = sc_1$ and $\ell_0 = \ell_1$.
(ii) For any ℓ, b, there exist sc, ℓ' such that $b\ {}^{\ell}\!\searrow_{\ell'} sc$. If $b\ {}^{\ell}\!\searrow_{\ell_0} sc_0$ and $b\ {}^{\ell}\!\searrow_{\ell_1} sc_1$, then $sc_0 = sc_1$ and $\ell_0 = \ell_1$.
(iii) For any ℓ, s, there exist sc, ℓ' such that $s\ {}^{\ell}\!\searrow_{\ell'} sc$. If $s\ {}^{\ell}\!\searrow_{\ell_0} sc_0$ and $s\ {}^{\ell}\!\searrow_{\ell_1} sc_1$, then $sc_0 = sc_1$ and $\ell_0 = \ell_1$.

Lemma 10 (Domain of compiled code)
(i) If $a\ {}^{\ell}\!\searrow_{\ell'} sc$, then $\mathrm{dom}(sc) = [\ell, \ell')$.
(ii) If $b\ {}^{\ell}\!\searrow_{\ell'} sc$, then $\mathrm{dom}(sc) = [\ell, \ell')$.
(iii) If $s\ {}^{\ell}\!\searrow_{\ell'} sc$, then $\mathrm{dom}(sc) = [\ell, \ell')$.

That compilation does not alter the meaning of an expression or statement is demonstrated by the facts that While evaluations are preserved and SPush evaluations are reflected by it. We must however take into account the fact a compiled While expression or statement is intended to be entered from its beginning-point.

Theorem 12 (Preservation of evaluations by compilation)
(i) If $a\ {}^{\ell}\!\searrow_{\ell'} sc$, then $(\ell, zs, \sigma) \succ sc \rightarrow (\ell', [\![a]\!]\sigma :: zs, \sigma)$.
(ii) If $b\ {}^{\ell}\!\searrow_{\ell'} sc$, then $(\ell, zs, \sigma) \succ sc \rightarrow (\ell', [\![b]\!]\sigma :: zs, \sigma)$.
(iii) If $s\ {}^{\ell}\!\searrow_{\ell'} sc$ and $\sigma \succ s \rightarrow \sigma'$, then $(\ell, zs, \sigma) \succ sc \rightarrow (\ell', zs, \sigma')$.

Proof. By induction on the structure of a or b or the derivation of $\sigma \succ s \rightarrow \sigma'$. $\square$

Theorem 13 (Reflection of evaluations by compilation)
(i) If $a\,{}^{\ell}\!\searrow_{\ell'} sc$ *and* $(\ell, zs, \sigma) \succ sc \rightarrow (\ell'', zs', \sigma')$, *then* $\ell'' = \ell'$, $zs' = [\![a]\!]\sigma :: zs$ *and* $\sigma' = \sigma$.
(ii) If $b\,{}^{\ell}\!\searrow_{\ell'} sc$ *and* $(\ell, zs, \sigma) \succ sc \rightarrow (\ell'', zs', \sigma')$, *then* $\ell'' = \ell'$, $zs' = [\![b]\!]\sigma :: zs$ *and* $\sigma' = \sigma$.
(iii) If $s\,{}^{\ell}\!\searrow_{\ell'} sc$ *and* $(\ell, zs, \sigma) \succ sc \rightarrow (\ell'', zs', \sigma')$, *then* $\ell'' = \ell'$, $zs' = zs$ *and* $\sigma \succ s \rightarrow \sigma'$.

Proof. By induction on the structure of a, b or s and subordinate induction on the derivation of $(\ell, zs, \sigma) \succ sc \rightarrow (\ell'', zs', \sigma')$. $\square$

It is easy to show that compilation preserves derivable WHILE Hoare triples (in a suitable format that takes into account that a WHILE statement proof assumes. But one can also give a constructive proof: a proof by defining a compositional translation of WHILE program proofs to SPUSH program proofs, i.e., a proof compilation function.

Theorem 14 (Preservation of derivable Hoare triples)
(i) If $a\,{}^{\ell}\!\searrow_{\ell'} sc$ *and* P *is a* WHILE *assertion, then* $\{pc = \ell \wedge st = zs \wedge P\}\, sc\, \{pc = \ell' \wedge st = a :: zs \wedge P\}$.
(ii) If $b\,{}^{\ell}\!\searrow_{\ell'} sc$ *and* P *is a* WHILE *assertion, then* $\{pc = \ell \wedge st = zs \wedge P\}\, sc\, \{pc = \ell' \wedge st = b :: zs \wedge P\}$.
(iii) If $s\,{}^{\ell}\!\searrow_{\ell'} sc$ *and* $\{P\}\, s\, \{Q\}$, *then* $\{pc = \ell \wedge st = zs \wedge P\}\, sc\, \{pc = \ell' \wedge st = zs \wedge Q\}$.

Proof. Non-constructive proof is straightforward from soundness of the Hoare logic of WHILE, reflection of evaluations by compilation and completeness of the Hoare logic of SPUSH.

Constructive proof is by induction on the structure of a or b or the derivation of $\{P\}\, s\, \{Q\}$. We will construct the derivation tree for sc in the compositional logic. Note that we will omit explicitly invoking the rule of consequence, since its invocation should be obvious from the context.

(i) The proof is by induction on the structure of a. We will make use of the fact that P or Q, being high-level assertions, do not include constants pc or st. We have the following cases

- $a = n$.
  Then $sc = (\ell, \mathsf{push}\ n)$ and $\ell' = \ell + 1$. We have the following derivation:

$$\frac{\left\{ \begin{array}{l} (pc = \ell \wedge n :: st = n :: zs \wedge P) \\ \vee\ (pc \neq \ell \wedge st = n :: zs \wedge P) \end{array} \right\} (\ell, \mathsf{push}\ n)\, \{pc = \ell + 1 \wedge st = n :: zs \wedge P\}}{\{pc = \ell \wedge st = zs \wedge P\}\, (\ell, \mathsf{push}\ n)\, \{pc = \ell + 1 \wedge st = n :: zs \wedge P\}}$$

- $a = x$.
  Then $sc = (\ell, \mathsf{load}\ x)$ and $\ell' = \ell + 1$. We ave the following derivation:

$$\frac{\left\{ \begin{array}{l} (pc = \ell \wedge x :: st = x :: zs \wedge P) \\ \vee\ (pc \neq \ell \wedge st = x :: zs \wedge P) \end{array} \right\} (\ell, \mathsf{load}\ x)\, \{pc = \ell + 1 \wedge st = x :: zs \wedge P\}}{\{pc = \ell \wedge st = zs \wedge P\}\, (\ell, \mathsf{load}\ x)\, \{pc = \ell + 1 \wedge st = x :: zs \wedge P\}}$$

- $a = a_0 + a_1$.
  Then there are $\ell_0, \ell_1, sc_0, sc_1$ such that $a_0 \,{}^{\ell}\!\searrow_{\ell_0} sc_0$, $a_1 \,{}^{\ell_0}\!\searrow_{\ell_1} sc_1$, $sc = (sc_0 \oplus sc_1) \oplus \mathsf{add}$ and $\ell' = \ell_1 + 1$.

  Let

$$
\begin{array}{rcl}
I & = & (pc = \ell \wedge st = zs \wedge P) \vee (pc = \ell_1 \wedge st = a_1 :: a_0 :: zs \wedge P) \\
 & \vee & (pc = \ell' \wedge st = a_0 + a_1 :: zs \wedge P) \\
I_1 & = & (pc = \ell \wedge st = zs \wedge P) \vee (pc = \ell_0 \wedge st = a_0 :: zs \wedge P) \\
 & \vee & (pc = \ell_1 \wedge st = a_1 :: a_0 :: zs \wedge P)
\end{array}
$$

  We get the following derivation:

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots \text{ Ind. Hypot.}}{\{pc = \ell \wedge st = zs \wedge P\}\, sc_0 \,\{pc = \ell_0 \wedge st = a_0 :: zs \wedge P\}}}{\{pc \in [\ell, \ell_0) \wedge I_1\}\, sc_0 \,\{I_1\}} \quad \dfrac{\dfrac{\vdots \text{ Ind. Hypot.}}{\{pc = \ell_0 \wedge st = a_0 :: zs \wedge P\}\, sc_1 \,\{pc = \ell_1 \wedge st = a_1 :: a_0 :: zs \wedge P\}}}{\{pc \in [\ell_0, \ell_1) \wedge I_1\}\, sc_1 \,\{I_1\}}}{\{I_1\}\, sc_0 \oplus sc_1 \,\{pc \notin [\ell, \ell_1) \wedge I_1\}}}{\{pc \in [\ell, \ell_1) \wedge I\}\, sc_0 \oplus sc_1 \,\{I\}} \quad (1)}{\{I\}\, (sc_0 \oplus sc_1) \oplus (\ell_1, \mathsf{add})\, \{I \wedge pc \notin [\ell, \ell')\}}}{\{pc = \ell \wedge st = zs \wedge P\}\, (sc_0 \oplus sc_1) \oplus (\ell_1, \mathsf{add})\, \{pc = \ell' \wedge st = a_0 + a_1 :: zs \wedge P\}}
$$

  $(1) =$

$$
\dfrac{\left\{ \begin{array}{ll} & (pc = \ell_1 \wedge \exists z_0, z_1 \in \mathbb{Z}, w \in (\mathbb{Z} + \mathbb{B})^* .\, st = z_0 :: z_1 :: w \\ & \quad \wedge \ell' = \ell' \wedge z_0 + z_1 :: w = a_0 + a_1 :: zs \wedge P) \\ \vee & (pc \neq \ell_1 \wedge pc = \ell' \wedge st = a_0 + a_1 :: zs \wedge P) \end{array} \right\} (\ell_1, \mathsf{add}) \,\{\ pc = \ell' \wedge st = a_0 + a_1 :: zs \wedge P\ \}}{\{pc = \ell_1 \wedge I\}\, (\ell_1, \mathsf{add})\, \{I\}}
$$

(ii)

- $b_0 = b_1$
  Analogous to $a_0 + a_1$ case.

(iii) The proof is by induction on s. Assume that $s \,{}^{\ell}\!\searrow_{\ell'} sc$. We have the following cases.

- The derivation of $\{P\}\, s\, \{Q\}$ is

$$
\dfrac{}{\{Q[a/x]\}\, x := a\, \{Q\}}
$$

Then there are ℓ_0, sc_a such that $a \,{}^{\ell}\!\searrow_{\ell_0} sc_a$, $sc = sc_a \oplus (\ell_0, \mathsf{store}\ x)$ and $\ell' = \ell_0 + 1$.

Let

$$\begin{array}{rcl} I & = & (pc = \ell \wedge st = zs \wedge P[a/x]) \vee (pc = \ell' \wedge st = zs \wedge P) \\ & \vee & (pc = \ell_1 \wedge st = a :: zs \wedge P[a/x]) \end{array}$$

We have the following derivation:

$$\dfrac{\dfrac{\dfrac{\{pc = \ell \wedge st = zs \wedge P[a/x]\}\, sc_a \,\{pc = \ell_1 \wedge st = a :: zs \wedge P[a/x]\}}{\{pc \in [\ell, \ell') \wedge I\}\, sc_a \,\{I\} \qquad (1)}}{\{I\}\, sc_a \oplus (\ell_0, \mathsf{store}\ x)\, \{pc \notin [\ell, \ell') \wedge I\}}}{\{pc = \ell \wedge st = zs \wedge P[a/x]\}\, sc_a \oplus (\ell_0, \mathsf{store}\ x)\, \{pc = \ell' \wedge st = zs \wedge P\}}$$

where (1) =

$$\dfrac{\dfrac{\left\{\begin{array}{ll} & (pc = \ell_0 \wedge \exists z \in \mathbb{Z}, w \in (\mathbb{Z} + \mathbb{B})^*.\, st = z :: w \\ & \wedge \ell' = \ell' \wedge w = zs \wedge P[z/x]) \\ \vee & (pc \neq \ell_0 \wedge pc = \ell' \wedge st = zs \wedge P) \end{array}\right\} (\ell_0, \mathsf{store}\ x)\ \{\ pc = \ell' \wedge st = zs \wedge P\ \}}{\{pc = \ell_0 \wedge st = a :: zs \wedge P[a/x]\}\, (\ell_0, \mathsf{store}\ x)\, \{pc = \ell' \wedge st = zs \wedge P\}}}{\{pc \in [\ell, \ell') \wedge I\}\, (\ell_0, \mathsf{store}\ x)\, \{I\}}$$

- The derivation of $\{P\}\, s\, \{Q\}$ is

$$\dfrac{}{\{P\}\, \mathsf{skip}\, \{P\}}$$

Then $sc = \mathbf{0}$ and $\ell' = \ell$. We have the derivation

$$\dfrac{}{\{pc = \ell \wedge st = zs \wedge P\}\, \mathbf{0}\, \{pc = \ell \wedge st = zs \wedge P\}}$$

- The derivation of $\{P\}\, s\, \{Q\}$ is

$$\dfrac{\{P\}\, s_0\, \{R\} \quad \{R\}\, s_1\, \{Q\}}{\{P\}\, s_0; s_1\, \{Q\}}$$

Then there are ℓ'', sc_0, sc_1 such that $s_0 \,{}^{\ell}\!\searrow_{\ell''} sc_0$, $s_1 \,{}^{\ell''}\!\searrow_{\ell'} sc_1$ and $sc = sc_0 \oplus sc_1$.
Let

$$\begin{array}{rcl} I & = & (pc = \ell \wedge st = zs \wedge P) \vee (pc = \ell'' \wedge st = zs \wedge R) \\ & \vee & (pc = \ell' \wedge st = zs \wedge Q) \end{array}$$

We have the following derivation:

$$
\dfrac{
\dfrac{
\dfrac{\vdots\ \text{Ind. Hypot.}}{\{pc = \ell \wedge st = zs \wedge P\}\, sc_0\, \{pc = \ell'' \wedge st = zs \wedge R\}}
}{\{pc \in [\ell, \ell'') \wedge I\}\, sc_0\, \{I\}}
\qquad
\dfrac{
\dfrac{\vdots\ \text{Ind. Hypot.}}{\{pc = \ell'' \wedge st = zs \wedge R\}\, sc_1\, \{pc = \ell' \wedge st = zs \wedge Q\}}
}{\{pc \in [\ell'', \ell') \wedge I\}\, sc_1\, \{I\}}
}{
\dfrac{\{I\}\, sc_0 \oplus sc_1\, \{pc \notin [\ell, \ell') \wedge I\}}{\{pc = \ell \wedge st = zs \wedge P\}\, sc_0 \oplus sc_1\, \{pc = \ell' \wedge st = zs \wedge Q\}}
}
$$

- The derivation of $\{P\}\, s\, \{Q\}$ is

$$
\dfrac{\begin{array}{c}\vdots\\ \{b \wedge P\}\, s_t\, \{Q\}\end{array} \qquad \begin{array}{c}\vdots\\ \{\neg b \wedge P\}\, s_f\, \{Q\}\end{array}}{\{P\}\, \textsf{if}\ b\ \textsf{then}\ s_t\ \textsf{else}\ s_f\, \{Q\}}
$$

Then there are $\ell'', \ell''', sc_b, sc_t, sc_f$ such that $b\ {}^{\ell}\!\searrow_{\ell''} sc_b$, $s_t\ {}^{\ell''+1}\!\searrow_{\ell'''} sc_t$, $s_f\ {}^{\ell'''+1}\!\searrow_{\ell'} sc_f$ and $sc = \overbrace{(sc_b \oplus (\ell'', \textsf{gotoF}\ \ell''' + 1))}^{sc_1} \oplus \overbrace{(\underbrace{(sc_t \oplus (\ell''', \textsf{goto}\ \ell'))}_{sc_3} \oplus sc_f)}^{sc_2}$.

Let

$$
\begin{array}{rcl}
I & = & (pc = \ell \wedge st = zs \wedge P) \vee (pc = \ell'' + 1 \wedge st = zs \wedge b \wedge P) \\
& \vee & (pc = \ell' \wedge st = zs \wedge Q) \\
I_1 & = & (pc = \ell \wedge st = zs \wedge P) \vee (pc = \ell'' \wedge st = b :: zs \wedge P) \\
& \vee & (pc = \ell'' + 1 \wedge st = zs \wedge b \wedge P) \vee (pc = \ell''' + 1 \wedge st = zs \wedge \neg b \wedge P) \\
I_2 & = & (pc = \ell'' + 1 \wedge st = zs \wedge b \wedge P) \vee (pc = \ell''' + 1 \wedge st = zs \wedge \neg b \wedge P) \\
& \vee & (pc = \ell' \wedge st = zs \wedge Q) \\
I_3 & = & (pc = \ell'' + 1 \wedge st = zs \wedge b \wedge P) \vee (pc = \ell''' \wedge st = zs \wedge Q) \\
& \vee & (pc = \ell' \wedge st = zs \wedge Q)
\end{array}
$$

$$
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{\vdots\ \text{(ii)}}{\{pc = \ell \wedge st = zs \wedge P\}\, sc_b\, \{pc = \ell'' \wedge st = b :: zs \wedge P\}}
}{\{pc \in [\ell, \ell'') \wedge I_1\}\, sc_b\, \{I_1\}}\ (1)
}{\{I_1\}\, sc_1\, \{pc \notin [\ell, \ell'' + 1) \wedge I_1\}}
}{\{pc \in [\ell, \ell'' + 1) \wedge I\}\, sc_1\, \{I\}}\ (2)
}{
\dfrac{\{I\}\, sc\, \{pc \notin [\ell, \ell') \wedge I\}}{\{pc = \ell \wedge st = zs \wedge P\}\, sc\, \{pc = \ell' \wedge st = zs \wedge Q\}}
}
$$

where
(1) =

$$\dfrac{\left\{\begin{array}{ll} & (pc = \ell'' \quad \wedge((\ell''' + 1 \neq \ell'' \quad \wedge((\exists w \in (\mathbb{Z}+\mathbb{B})^*.\, st = \mathrm{tt} :: w \\ & \quad ((\ell'' + 1 = \ell'' + 1 \wedge b) \vee (\ell'' + 1 = \ell''' + 1 \wedge \neg b)) \wedge w = zs \wedge P) \\ & \quad \vee(\exists w \in (\mathbb{Z}+\mathbb{B})^*.\, st = \mathrm{ff} :: w \\ & \quad ((\ell''' + 1 = \ell'' + 1 \wedge b) \vee (\ell''' + 1 = \ell''' + 1 \wedge \neg b)) \wedge w = zs \wedge P))) \\ & \quad \vee \ldots \\ \vee & .. \end{array}\right\} \begin{array}{c} (\ell'', \textsf{gotoF}\ \ell''' + 1) \\ \{(pc = \ell'' + 1 \wedge b) \vee (pc = \ell''' + 1 \wedge \neg b)) \wedge st = zs \wedge P\} \end{array}}{\{pc = \ell'' \wedge I_1\}\,(\ell'', \textsf{gotoF}\ \ell')\,\{I_1\}}$$

and (2) =

$$\dfrac{\dfrac{}{\left\{\begin{array}{ll} & (pc = \ell''' \wedge ((\ell''' \neq \ell' \wedge \ell' = \ell' \wedge st = zs \wedge Q) \\ & \quad \vee \ell' = \ell''')) \\ \vee & (pc \neq \ell''' \wedge pc = \ell' \wedge st = zs \wedge Q) \end{array}\right\} (\ell''', \textsf{goto}\ \ell')\ \{\ pc = \ell' \wedge st = zs \wedge Q\ \}}}{\{pc = \ell''' \wedge I_3\}\,(\ell''', \textsf{goto}\ \ell')\,\{I_3\}}$$

$$\dfrac{\dfrac{\dfrac{\vdots\ \text{Ind. Hypot.}}{\{pc = \ell'' + 1 \wedge st = zs \wedge b \wedge P\}\, sc_t\, \{pc = \ell''' \wedge st = zs \wedge Q\}}}{\{pc \in [\ell'' + 1, \ell''') \wedge I_3\}\, sc_t\, \{I_3\}} \quad \{pc = \ell''' \wedge I_3\}\,(\ell''', \textsf{goto}\ \ell')\,\{I_3\}}{\{I_3\}\, sc_3\, \{pc \in [\ell'' + 1, \ell''' + 1) \wedge I_3\}}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\{I_3\}\, sc_3\, \{pc \in [\ell'' + 1, \ell''' + 1) \wedge I_3\}}}{\{pc \in [\ell'' + 1, \ell''' + 1) \wedge I_2\}\, sc_3\, \{I_2\}} \quad \dfrac{\dfrac{\vdots\ \text{Ind. Hypot.}}{\{pc = \ell''' + 1 \wedge st = zs \wedge \neg b \wedge P\}\, sc_f\, \{pc = \ell' \wedge st = zs \wedge Q\}}}{\{pc \in [\ell''' + 1, \ell') \wedge I_2\}\, sc_f\, \{I_2\}}}{\{I_2\}\, sc_2\, \{pc \notin [\ell'' + 1, \ell') \wedge I_2\}}}{\{pc \in [\ell'' + 1, \ell') \wedge I)\}\, sc_2\, \{I\}}$$

- The derivation of $\{P\}\, s\, \{Q\}$ is

$$\dfrac{\begin{array}{c}\vdots \\ \{b \wedge P\}\, s_t\, \{P\}\end{array}}{\{P\}\ \textsf{while}\ b\ \textsf{do}\ s_t\ \{\neg b \wedge P\}}$$

Then there are $\ell'', \ell''', sc_b, sc_t$ such that $b\ {}^{\ell}\!\searrow_{\ell''} sc_b$, $s_t\ {}^{\ell''+1}\!\searrow_{\ell'''} sc_t$ and $sc = \overbrace{(sc_b \oplus (\ell'', \textsf{gotoF}\ \ell'))}^{sc_1} \oplus \overbrace{(sc_t \oplus (\ell''', \textsf{goto}\ \ell))}^{sc_2}$ and $\ell' = \ell''' + 1$.
Let

$$\begin{array}{rcl} I & = & (pc = \ell \wedge st = zs \wedge P) \vee (pc = \ell'' + 1 \wedge st = zs \wedge b \wedge P) \\ & \vee & (pc = \ell' \wedge st = zs \wedge \neg b \wedge P) \\ I_1 & = & (pc = \ell \wedge st = zs \wedge P) \vee (pc = \ell'' \wedge st = b :: zs \wedge P) \\ & \vee & (pc = \ell'' + 1 \wedge st = zs \wedge b \wedge P) \vee (pc = \ell' \wedge st = zs \wedge \neg b \wedge P) \\ I_2 & = & (pc = \ell'' + 1 \wedge st = zs \wedge b \wedge P) \vee (pc = \ell''' \wedge st = zs \wedge P) \\ & \vee & (pc = \ell' \wedge st = zs \wedge \neg b \wedge P) \end{array}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots\ \text{(ii)}}{\{pc = \ell \wedge st = zs \wedge P\}\, sc_b\, \{pc = \ell'' \wedge st = b :: zs \wedge P\}}}{\{pc \in [\ell, \ell'') \wedge I_1\}\, sc_b\, \{I_1\}} \quad (1)}{\{I_1\}\, sc_1\, \{pc \notin [\ell, \ell''+1) \wedge I_1\}}}{\{pc \in [\ell, \ell''+1) \wedge I\}\, sc_1\, \{I\}} \quad \dfrac{\dfrac{\dfrac{\dfrac{\vdots\ \text{Ind. Hypot.}}{\{pc = \ell''+1 \wedge st = zs \wedge b \wedge P\}\, sc_t\, \{pc = \ell''' \wedge st = zs \wedge P\}}}{\{pc \in [\ell''+1, \ell''') \wedge I_2\}\, sc_t\, \{I_2\}} \quad (2)}{\{I_2\}\, sc_2\, \{pc \notin [\ell''+1, \ell') \wedge I_2\}}}{\{pc \in [\ell''+1, \ell') \wedge I\}\, sc_2\, \{I\}}}{\dfrac{\{I\}\, sc\, \{pc \notin [\ell, \ell') \wedge I\}}{\{pc = \ell \wedge st = zs \wedge P\}\, sc\, \{pc = \ell' \wedge st = zs \wedge \neg b \wedge P\}}}$$

where

(1) =

$$\dfrac{\left\{\begin{array}{ll} & (pc = \ell'' \quad \wedge((\ell' \neq \ell'' \quad \wedge((\exists w \in (\mathbb{Z}+\mathbb{B})^*.\, st = \mathrm{tt} :: w \\ & \qquad ((\ell''+1 = \ell''+1 \wedge b) \vee (\ell''+1 = \ell' \wedge \neg b)) \wedge w = zs \wedge P) \\ & \qquad\qquad \vee(\exists w \in (\mathbb{Z}+\mathbb{B})^*.\, st = \mathrm{ff} :: w \\ & \qquad ((\ell' = \ell''+1 \wedge b) \vee (\ell' = \ell' \wedge \neg b)) \wedge w = zs \wedge P))) \\ & \qquad \vee ... \\ \vee & ... \end{array}\right\} \\ (\ell'', \mathsf{gotoF}\ \ell') \\ \{(pc = \ell''+1 \wedge b) \vee (pc = \ell' \wedge \neg b)) \wedge st = zs \wedge P\}}{\{pc = \ell'' \wedge I_1\}\, (\ell'', \mathsf{gotoF}\ \ell')\, \{I_1\}}$$

(2) =

$$\dfrac{\dfrac{}{\left\{\begin{array}{ll} & (pc = \ell''' \wedge ((\ell''' \neq \ell \wedge \ell = \ell \wedge st = zs \wedge P) \vee \ell''' = \ell)) \\ \vee & (pc \neq \ell''' \wedge pc = \ell' \wedge st = zs \wedge P) \end{array}\right\} (\ell''', \mathsf{goto}\ \ell)\ \{\ pc = \ell' \wedge st = zs \wedge P\ \}}}{\{pc = \ell''' \wedge I_2\}\, (\ell''', \mathsf{goto}\ \ell)\, \{I_2\}}$$

□

3.4.1 Example

As a simple example of compilation we present a While factorial program together with its proof, and then the target SGoto program with its proof. The factorial program in While is $S =_{\mathrm{df}}$ $\mathsf{while}\ x < n\ \mathsf{do}\ (x := x+1; s := s*x)$. For this program, we have the following Hoare proof (we refrain here from explicitly spelling out the side conditions of consequence inferences, these are obvious from the context).

$$
\dfrac{
  \dfrac{
    \dfrac{}{\{x+1\le n \wedge s*(x+1)=(x+1)!\}\; x:=x+1\; \{x\le n \wedge s*x=x!\}}
  }{\{x<n \wedge s=x!\}\; x:=x+1\; \{x\le n \wedge s*x=x!\}}
  \qquad
  \dfrac{}{\{x\le n \wedge s*x=x!\}\; s:=s*x\; \{s=x! \wedge x\le n\}}
}{
  \dfrac{
    \dfrac{
      \dfrac{\{x<n\wedge s=x!\}\; x:=x+1; s:=s*x\; \{x\le n\wedge s=x!\}}{\{x<n\wedge x\le n\wedge s=x!\}\; x:=x+1; s:=s*x\; \{x\le n\wedge s=x!\}}
    }{\{x\le n\wedge s=x!\}\; S\; \{x\not< n\wedge x\le n\wedge s=x!\}}
  }{\{n\ge 0\wedge x=0\wedge s=1\}\; S\; \{x=n\wedge s=n!\}}
}
$$

The compilation function gives us the following SPush program (sc_0).

1	load x			
2	load n	sc_3	sc_1	
3	less			
4	gotoF 14			
5	load x			
6	push 1	sc_4		
7	add			sc_0
8	store x			
9	load s		sc_2	
10	load x	sc_5		
11	mult			
12	store s			
13	goto 1			
14				

To present its proof we introduce the notations

$$
\begin{array}{lcl}
I_{1'} & =_{\mathrm{df}} & pc=1\wedge st=zs\wedge n\ge 0\wedge x=0\wedge s=1\\
I_{1} & =_{\mathrm{df}} & pc=1\wedge st=zs\wedge x\le n\wedge s=x!\\
I_{2} & =_{\mathrm{df}} & pc=2\wedge st=x::zs\wedge x\le n\wedge s=x!\\
I_{3} & =_{\mathrm{df}} & pc=3\wedge st=n::x::zs\wedge x\le n\wedge s=x!\\
I_{4} & =_{\mathrm{df}} & pc=4\wedge st=x<n::zs\wedge x\le n\wedge s=x!\\
I_{5} & =_{\mathrm{df}} & pc=5\wedge st=zs\wedge x<n\wedge s=x!\\
I_{6} & =_{\mathrm{df}} & pc=6\wedge st=x::zs\wedge x<n\wedge s=x!\\
I_{7} & =_{\mathrm{df}} & pc=7\wedge st=1::x::zs\wedge x<n\wedge s=x!\\
I_{8'} & =_{\mathrm{df}} & pc=8\wedge st=x+1::zs\wedge x<n\wedge s=x!\\
I_{8} & =_{\mathrm{df}} & pc=8\wedge st=x+1::zs\wedge x+1\le n\wedge s*(x+1)=(x+1)!\\
I_{9} & =_{\mathrm{df}} & pc=9\wedge st=zs\wedge x\le n\wedge s*x=x!\\
I_{10} & =_{\mathrm{df}} & pc=10\wedge st=s::zs\wedge x\le n\wedge s*x=x!\\
I_{11} & =_{\mathrm{df}} & pc=11\wedge st=x::s::zs\wedge x\le n\wedge s*x=x!\\
I_{12} & =_{\mathrm{df}} & pc=12\wedge st=s*x::zs\wedge x\le n\wedge s*x=x!\\
I_{13} & =_{\mathrm{df}} & pc=13\wedge st=zs\wedge x\le n\wedge s=x!\\
I_{14} & =_{\mathrm{df}} & pc=14\wedge st=zs\wedge x\not< n\wedge x\le n\wedge s=x!\\
I_{14'} & =_{\mathrm{df}} & pc=14\wedge st=zs\wedge x=n\wedge s=n!.
\end{array}
$$

We will use the shorthand notation $I_{i,\dots,j}$ to denote the disjunction $I_i \vee \ldots \vee I_j$.

The proof for the PUSH program is the following (the Hoare triples corresponding to those in the WHILE version are highlighted):

$$\{J_9\}\,9\,\{I_{10}\} \qquad \{J_{10}\}\,10\,\{I_{11}\}$$

$$\{pc = 9 \wedge I_{9,10,11}\}\,9\,\{I_{9,20,11}\} \qquad \{pc = 10 \wedge I_{9,10,11}\}\,10\,\{I_{9,10,11}\}$$

$$\{I_{9,10,11}\}\,9 \oplus 10\,\{pc \notin [9,11) \wedge I_{9,10,11}\}$$

$$\{pc \in [9,11) \wedge I_{9,11,12}\}\,9 \oplus 10\,\{I_{9,11,12}\}$$

$$\{J_{11}\}\,11\,\{I_{12}\}$$

$$\{pc = 11 \wedge I_{9,11,12}\}\,11\,\{I_{9,11,12}\}$$

$$\{I_{9,11,12}\}\,9 \oplus 10 \oplus 11\,\{pc \notin [9,12) \wedge I_{9,11,12}\}$$

$$\{pc \in [9,12) \wedge I_{9,12,13}\}\,9 \oplus 10 \oplus 11\,\{I_{9,12,13}\}$$

$$\{J_5\}\,5\,\{I_6\} \qquad \{J_6\}\,6\,\{I_7\}$$

$$\{pc = 5 \wedge I_{5,6,7}\}\,5\,\{I_{5,6,7}\} \qquad \{pc \in 6 \wedge I_{5,6,7}\}\,6\,\{I_{5,6,7}\}$$

$$\{I_{5,6,7}\}\,5 \oplus 6\,\{pc \notin [5,7) \wedge I_{5,6,7}\}$$

$$\{pc \in [5,7) \wedge I_{5,7,8}\}\,5 \oplus 6\,\{I_{5,7,8}\}$$

$$\{J_7\}\,7\,\{I_{8'}\}$$

$$\{J_7\}\,7\,\{I_8\}$$

$$\{pc = 7 \wedge I_{5,7,8}\}\,7\,\{I_{5,7,8}\}$$

$$\{I_{5,7,8}\}\,5 \oplus 6 \oplus 7\,\{pc \notin [5,8) \wedge I_{5,7,8}\}$$

$$\{pc \in [5,9) \wedge I_{5,8,9}\}\,5 \oplus 6 \oplus 7\,\{I_{5,8,9}\}$$

$$\{J_{12}\}\,12\,\{I_{13}\}$$

$$\{pc = 12 \wedge I_{9,12,13}\}\,12\,\{I_{9,12,13}\}$$

$$\{I_{9,12,13}\}\,sc_5\,\{pc \notin [9,13) \wedge I_{9,12,13}\}$$

$$\{pc \in [9,13) \wedge I_{5,9,13}\}\,sc_5\,\{I_{5,9,13}\}$$

$$\{J_8\}\,8\,\{I_9\}$$

$$\{pc = 8 \wedge I_{5,8,9}\}\,8\,\{I_{5,8,9}\}$$

$$\{I_{5,8,9}\}\,sc_4\,\{pc \notin [5,9) \wedge I_{5,8,9}\}$$

$$\{pc \in [5,9) \wedge I_{5,9,13}\}\,sc_4\,\{I_{5,9,13}\}$$

$$\{I_{5,9,13}\}\,sc_4 \oplus sc_5\,\{pc \notin [5,13) \wedge I_{5,9,13}\}$$

$$\{pc \in [5,13) \wedge I_{1,5,13}\}\,sc_4 \oplus sc_5\,\{I_{1,5,13}\}$$

$$\{J_1\}\,1\,\{I_2\} \qquad \{J_2\}\,2\,\{I_3\}$$

$$\{pc = 1 \wedge I_{1,2,3}\}\,1\,\{I_{1,2,3}\} \qquad \{pc = 2 \wedge I_{1,2,3}\}\,2\,\{I_{1,2,3}\}$$

$$\{I_{1,2,3}\}\,1 \oplus 2\,\{pc \notin [1,3) \wedge I_{1,2,3}\}$$

$$\{pc \in [1,3) \wedge I_{1,3,4}\}\,1 \oplus 2\,\{I_{1,3,4}\}$$

$$\{J_3\}\,3\,\{I_4\}$$

$$\{pc = 3 \wedge I_{1,3,4}\}\,3\,\{I_{1,3,4}\}$$

$$\{I_{1,3,4}\}\,sc_3\,\{pc \notin [1,4) \wedge I_{1,3,4}\}$$

$$\{pc \in [1,4) \wedge I_{1,4,5,14}\}\,sc_3\,\{I_{1,4,5,14}\}$$

$$\{J_{13}\}\,\mathsf{goto}\ 1\,\{I_1\}$$

$$\{pc = 13 \wedge I_{1,5,13}\}\,\mathsf{goto}\ 1\,\{I_{1,5,13}\}$$

$$\{I_{1,5,13}\}\,sc_2\,\{pc \notin [5,14) \wedge I_{1,5,13}\}$$

$$\{pc \in [5,14) \wedge I_{1,5,14}\}\,sc_2\,\{I_{1,5,14}\}$$

$$\{J_4\}\,\mathsf{gotoF}\ 14\,\{I_{5,14}\}$$

$$\{pc = 4 \wedge I_{1,4,5,14}\}\,\mathsf{gotoF}\ 14\,\{I_{1,4,5,14}\}$$

$$\{I_{1,4,5,14}\}\,sc_1\,\{pc \notin [1,5) \wedge I_{1,4,5,14}\}$$

$$\{pc \in [1,5) \wedge I_{1,5,14}\}\,sc_1\,\{I_{1,5,14}\}$$

$$\{I_{1,5,14}\}\,sc_0\,\{pc \notin [1,14) \wedge I_{1,5,14}\}$$

$$\{I_1\}\,sc_0\,\{I_{14}\}$$

$$\{I_{1''}\}\,sc_0\,\{I_{14}\}$$

$$\{I'_1\}\,sc_0\,\{I_{14'}\}$$

where

$$
\begin{array}{lll}
J_1 & =_{\rm df} & (pc = 1 \wedge I_2[2, x :: st/pc, st]) \vee (pc \neq 1 \wedge I_2) \\
J_2 & =_{\rm df} & (pc = 2 \wedge I_3[3, n :: st/pc, st]) \vee (pc \neq 2 \wedge I_3) \\
J_3 & =_{\rm df} & (pc = 3 \wedge \exists z_0, z_1, w.st = z_0 :: z_1 :: w \wedge I_4[4, z_1 < z_0 :: st/pc, st]) \vee (pc \neq 3 \wedge I_4) \\
J_4 & =_{\rm df} & (pc = 4 \wedge (14 \neq 4 \wedge (\exists w.st = \mathrm{tt} :: w \wedge I_{5,14}[5, w/pc, st]) \\
 & & \quad \vee(\exists w.st = \mathrm{ff} :: w \wedge I_{5,14}[14, w/pc, st])) \vee (14 = 4 \wedge ...)) \vee (pc \neq 4 \wedge I_{5,14}) \\
J_5 & =_{\rm df} & (pc = 5 \wedge I_6[6, x :: st/pc, st]) \vee (pc \neq 5 \wedge I_6) \\
J_6 & =_{\rm df} & (pc = 6 \wedge I_7[7, 1 :: st/pc, st]) \vee (pc \neq 6 \wedge I_7) \\
J_7 & =_{\rm df} & (pc = 7 \wedge \exists z_0, z_1, w.st = z_0 :: z_1 :: w \wedge I_8[8, z_1 + z_0 :: st/pc, st]) \vee (pc \neq 7 \wedge I_8) \\
J_8 & =_{\rm df} & (pc = 8 \wedge \exists z, w.(st = z :: w \wedge I_9[9, w, z/pc, st, x])) \vee (pc \neq 8 \wedge I_9) \\
J_9 & =_{\rm df} & (pc = 9 \wedge I_{10}[10, s :: st/pc, st]) \vee (pc \neq 9 \wedge I_{10}) \\
J_{10} & =_{\rm df} & (pc = 10 \wedge I_{11}[11, x :: st/pc, st]) \vee (pc \neq 10 \wedge I_{11}) \\
J_{11} & =_{\rm df} & (pc = 11 \wedge \exists z_0, z_1, w.st = z_0 :: z_1 :: w \wedge I_{12}[12, z_1 * z_0 :: st/pc, st]) \vee (pc \neq 11 \wedge I_{12}) \\
J_{12} & =_{\rm df} & (pc = 12 \wedge \exists z, w.(st = z :: w \wedge I_{13}[9, w, z/pc, st, s])) \vee (pc \neq 12 \wedge I_{13}) \\
J_{13} & =_{\rm df} & (pc = 13 \wedge (1 \neq 13 \wedge I_1[1/pc]) \vee 1 = 13) \vee (pc \neq 13 \wedge I_1)
\end{array}
$$

The example should explain the general idea of modularity of our logic: there is no need of global information for a judgement, but only the invariants for the entry and exit labels of the code at hand. As code is composed, we can eliminate the invariants of the intermediate entries and exits that are not required any more, so at the root of the tree, we only have the entry and exit invariants of the whole program, i.e., I_1 and $I_{14'}$.

From the example it is also obvious that the logic is much too verbose for practical PCC applications. For practical use, the logic could both be simplified (for example by assuming that code is always well-typed) and proof compression techniques applied, such as the ones described in [47]. Also, it would be practical to treat basic blocks rather than single instructions as elementary building blocks.

3.5 Related work

Hoare's original logic [30] was for While. After that, quite a few proposals were made for dealing with extensions of While, including both general and restricted jumps. The first in this line was a paper by Clint and Hoare, which uses conditional Hoare triples, establishing that an invariant has to hold at goto instructions, to guarantee the precondition on the label. The rule they give for conventional labeling makes certain assumptions about the form of the program: only one label per block, and no jumps *into* structures. This approach is extended in the paper by Kowaltowski [36], where each block is assumed to have an explicit label. De Bruijn [20] introduces both denotational semantics and a logic for a language with jumps. The logic is similar to that of Clint and Hoare, but instead of a natural deduction system for dealing with gotos, the logic takes a sequent calculus form so that the validity of a formula can be defined more directly. In the logic by Arbib and Alagić

[3], Hoare triples can have multiple postconditions. This reflects the fact that if there is a jump instruction in a compound statement, the statement has multiple exits.

Logics for low-level languages without phrase structure have only become a topic of active research with the advent of PCC, with Java bytecode and .NET CIL being the main motivators. The one very notable exception is Floyd's seminal paper on a logic of control-flow graphs [29], preceding Hoare's work. One of the first papers on Java bytecode logic is by Quigley [51]. It is based on decompilation, so the logic applies to programs that come from a fixed compiler that preserves the structure of the high-level program. Benton's logic for a PUSH-like stack-based language involves global contexts of label invariants as de Bruin's logic [14]. Bannwart and Müller's logic extends the work of Benton to a subset of Java bytecode with objects and methods [6].

The work of Huisman and Jacobs [31] describes a Hoare logic for Java that can handle exceptions as well. Schröder and Mossakowski discuss a systematic method for designing Hoare logics for languages with monadic side-effects, in particular exceptions [57, 58].

We based our semantics and logic on the implicit structure of finite unions. The same structure is used by Tan and Appel [63, 62], who study the same language. But instead of introducing a natural semantics for the structured version of the language, they proceed from a small-step semantics ideology. As a result, they arrive at a continuation-style Hoare logic explainable by Appel and McAllester's 'indexed model' [2], involving the notion of a label continuation being approximately true. Benton defines a similar logic for a stack-based language with a typing component ensuring that the stack is used safely [15].

We looked at translating proofs of WHILE programs into proofs of SPUSH programs. This is related to our earlier work [11], where we showed the preservation proof obligations in a similar setting.

There are number of formal deductive systems for showing different properties for bytecode like languages. They are all typically non-compositional and make use of a global context of label types or assertions, such that a type is associated with every instruction in a program. One of the first in this line was the type system by Stata and Abadi, which describes a Java bytecode verifier [61]. Kobayashi and Kirane [34], and Barthe and Rezk [10] use this approach for a type system for secure information flow analyses for sequential Java bytecode.

3.6 Conclusion

In this chapter, we showed that it is possible to obtain a compositional natural semantics and Hoare logic based simply on the structure of finite unions of pieces of code. The semantics and the logic have the desired metatheoretic properties: the semantics is equivalent to the standard small-step semantics and the logic is sound and complete. The semantic and logic descriptions are inherently no more complex

than those for a standard high-level language. Most of the perceived complexity of the descriptions do not stem from compositionality, but rather from the presence of the stack and the possibility of abnormal termination, which needs to be dealt with explicitly. A major difference from the work of Benton and Tan and Appel is that we avoid continuations and interpret Hoare triples in the standard way.

We believe that this work is relevant and interesting in the context of PCC, since finite unions are a natural construction in realistic situations where a larger piece of code would very typically arise as a sum of smaller pieces that are separately produced, often by different producers, and should then also be proved correct separately.

We also showed that translating a proof of a WHILE program into a proof of the corresponding SPUSH program is straightforward in the context of a non-optimizing compiler. The issue of proof transformation in the light of optimizing compilers will be tackled in the following chapters.

Looking at the example, it is obvious that the logic is very verbose and if taken literally, would not be effective in a PCC setting. There are several ways to remedy this issue. First of all, the logic can be significantly simplified by making the assumption that the code is type-safe, i.e. passes the bytecode verifier. In this case, potential sources of abrupt termination such as stack underruns, values of wrong types on the stack or branching instructions looping back on themselves can be ignored in the logic. It would also be very easy to compress the proofs, since in the proof tree the same assertions associated with a label ($pc = ... \wedge ...$) appear multiple times. Exploring these issues is however outside the scope of this work.

Chapter 4

Proof-preserving program transformations

4.1 Introduction

Proof-carrying code is based on the idea that in a security-critical code transmission setting, the code producer should provide some evidence that the program she distributes is safe and/or functionally correct. The code consumer would thus receive the program together with a certificate (proof) that attests that the program has the desired properties.

The code producer would typically use some (interactive) verification environment to prove her source program. The question is how to communicate the verification result to the code consumer who will not have access to the source code of the program. It is clear that there should be a mechanism to allow compilation of the program proof together with the program.

In the previous chapter we showed how the Hoare proof of a high-level program could be compiled into the proof of its compiled version. In a related line of work, we have shown how proof obligations computed for the source program are identical to those of the compiled program, thus making proof transformation trivial [11]. However, both of these approaches consider a non-optimizing compiler. As soon as optimizations or other program transformations take place, proof transformation becomes non-trivial—a valid proof of a program may not be valid for the optimized version of the same program.

A simple example illustrating this issue is given in Figure 4.1. In the left column, we have a program which, given suitable values for i, s, p and n, computes $c * n$ and c^n and saves the the results to s and p, respectively. The pre- and postcondition for the program are $\{s = 0 \land p = 1 \land i = 0 \land n \geq 0\}$ and $\{s = c * n \land p = c^n\}$ and a suitable loop invariant is $\{s = c * i \land p = c^i \land i \leq n\}$. If we now perform dead code elimination on the program assuming p to be dead at the end of the program, we obtain the new program given in Figure 4.1 b. It is obvious that we cannot state anything about p in the postcondition anymore. Consequently, the precondition and the loop invariant should also be relaxed.

a)

$\{s = 0 \wedge p = 1 \wedge i = 0 \wedge n \geq 0\}$
while $i < n$ do
$\{s = c * i \wedge p = c^i \wedge i \leq n\}$
    $s := s + c;$
    $p := p * c;$
    $i := i + 1;$
$\{s = c * n \wedge p = c^n\}$

b)

$\{s = 0 \wedge i = 0 \wedge n \geq 0\}$
while $i < n$ do
$\{s = c * i \wedge i \leq n\}$
    $s := s + c;$
    skip;
    $i := i + 1;$
$\{s = c * n\}$

Figure 4.1: Example of annotation transformations required for dead code elimination

a)

$\{i = 0 \wedge x = 0 \wedge n \geq 0\}$

$t = a + b;$
while $i < n$ do
$\{x := i * (a + b) \wedge i \leq n\}$
    $x := x + (a + b);$
    $i := i + 1;$
$\{x = n * (a + b)\}$

b)

$\{i = 0 \wedge x = 0 \wedge n \geq 0\}$
$t' = a + b;$
$t = t';$
while $i < n$ do
$\{x := i * (a + b) \wedge i \leq n \wedge t' = a + b\}$
    $x := x + t';$
    $i := i + 1;$
$\{x = n * (a + b)\}$

Figure 4.2: Annotation transformations required for common subexpression elimination

It might seem that the reason why the assertions for the program (and therefore also the proof) need to be transformed in the given example is that the transformation did not preserve the semantics of the program (while the vast majority of standard program transformations are semantics-preserving). But in fact the same issues apply to semantics-preserving optimizations. This can be shown on the example of common sub-expression elimination, given in Figure 4.2. The original program is shown in the left column. The pre- and postcondition for the program are $\{i = 0 \wedge x = 0 \wedge n \geq 0\}$ and $\{x = n * (a + b)\}$. A suitable loop invariant for proving the program correct is $x = i * (a + b) \wedge i \leq n$. The right column shows the program after common subexpression elimination has been applied. The given loop invariant is not strong enough for the transformed program any more, since there is no use of $(a + b)$ in the optimized loop. The transformation of the invariant into $x = i * t' \wedge i \leq n$ in accordance with the optimization is not sufficient either, since

it would be too weak to imply the postcondition. Instead, the invariant needs to be strengthened to $\{x = i * (a + b) \wedge i \leq n \wedge t' = a + b\}$, recording the knowledge that t' is equal to $a + b$ in the loop. What can be observed on this example is that while the *whole program* optimization is indeed semantics-preserving, the same can not be said about its *sub-statements*: for example $x := x + (a + b)$ is replaced by $x := x + t'$, which are obviously not semantically equivalent in general, but only when some additional assumptions can be made about the state. This is exactly the reason why the proof needs to be modified: these extra assumptions need to be recorded in the assertions.

These examples also make it clear that in general assertion transformation is not just strengthening (which seems to be a common misconception) nor weakening, but the assertions for the original and optimized program can be incomparable. In the dead code elimination example, *less* can be stated about the final result of the program after the optimization, so the corresponding assertion needs to be weakened. The same holds for the the loop invariant. Since less has to be assumed from the pre-state for the invariant to hold, the precondition can also be weakened. In the common subexpression elimination example, we saw that assertions needed to be strengthened. If we combined these two optimizations, the new assertions would be incomparable to the original ones.

In this chapter, we tackle exactly the problem of proof transformation in the context of program optimizations based on dataflow-analysis. Such optimizations are typically presented in an algorithmic manner, whereby the algorithms do not work directly on the phrase structure of the given program, but rather on an intermediate form such as its control-flow graph. This is a good way to go about optimizing programs, but it is not an ideal presentation of what is done, if the optimizations are required to have justifications that can be communicated.

Our approach is based on describing dataflow analyses declaratively as type systems, so that the result of a particular program's analysis is a type derivation. Analysis results are presented in terms of types ascribed to expressions and statements, certified by type derivations, and the transformation component carries out the optimizations licensed by these type derivations. It turns out that we can use the same type derivation as a guidance for automatically transforming not only the program, but also its proofs.

A simplified view of a PCC scenario with program optimization happening on the producer's side is given in Figure 4.3. We are concerned with the stages shown in the gray box—simultaneous transformation of both a program and its proof guided by a type derivation representing the result of analyzing the program.

Our work also shows that type systems are a compact and useful way of describing dataflow analyses and optimizations in general: they can explain them well in a declarative fashion (separating the issues of determining what counts as a valid analysis or optimization result and how to find one) and make soundness and improvement simple to prove by structural induction on type derivations. In fact, proof optimization works namely because of this: automatic proof transformations are a

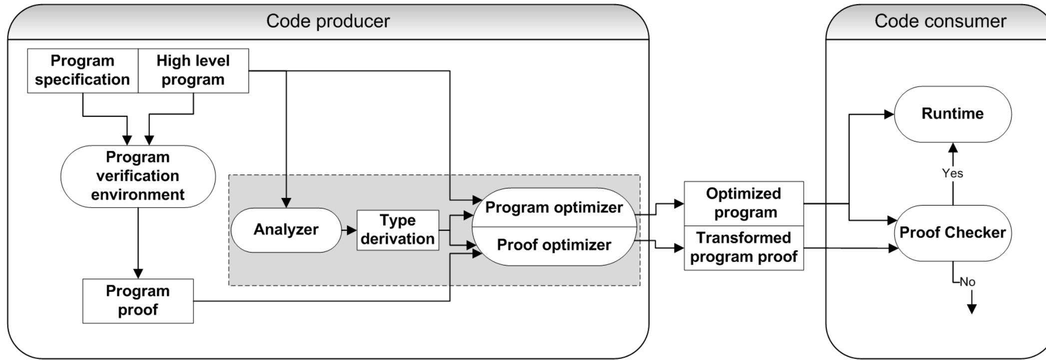


Figure 4.3: Proof optimization in PCC

formal version of the constructive content of these semantic arguments.

We demonstrate our approach on three program optimizations, namely dead code elimination, common subexpression elimination and partial redundancy elimination. All of the analyses are interesting from different aspects. Dead code elimination is an optimization that requires weakening of assertions in program proofs, which certificate-translation based approaches relying on assertion strengthening are not able to handle in a straightforward way. Common subexpression elimination requires two analyses (with the second analysis relying on the results of the first) for linking expression evaluation points to value reuse points and coordinated modifications of the program near both ends of such links, which seems to go against compositionality. We show how this can easily be overcome by a combined type system that reflects the combination of the two analyses. The scalability of our approach is demonstrated on partial redundancy elimination, which is a very subtle and complex optimization that changes the structure of the code by inserting nodes in control flow edges.

4.2 Dead code elimination

4.2.1 Type system for live variables analysis

We begin with type systems for dead code elimination and its underlying analysis, the live variables analysis. Discussing this analysis, we also comment on the general method for describing data-flow analyses as type systems.

We call a variable *live* at a program point, if there exists a path from that program point which (a) contains a useful use of the variable (by which we mean a use in an assignment to a variable that is live at the end of the assignment, or a use in an if- or while-guard) and (b) does not contain an assignment to the variable

before this use.[1] The corresponding live variables analysis determines, for each program point, which variables *may* be live at the program point. It is a *backward* analysis, starting from a set of variables that one wishes to consider live at the end of the program (at the top level, this would typically be **Var**: the final values of all variables are of interest).

The types and the subtyping relation of the type system corresponding to a data-flow analysis are the same as the underlying poset of the analysis, in this case the poset $(D, \leq) =_{\mathrm{df}} (\mathcal{P}(\mathbf{Var}), \supseteq)$. A state on a computation path has type $live \in D$, if some variable live in that state is not in *live*. The partial order is the opposite to the usual one for live variable analysis in order to get a natural subsumption rule (i.e. covariant in the posttype, contravariant in the pretype): from the point of subsumption, the natural analyses are "forward may" and "backward must" analyses; a backward may analysis is turned into a backward must analysis by reversing the partial order. The property specified by a type is negated, because the analysis is backward. (Let us recall that an analysis is a must analysis, if we require a property to be satisfied by all paths coming to a program point. For the may analysis, at least one path must have that property.)

A typing judgement for an arithmetic expression is of the form $a : live \longrightarrow live'$, where the pretype *live* and the posttype $live'$ are in each case elements of D; for boolean expressions and statements the judgements are similar. Generally, the intended meaning of a typing judgement is that, if the property specified by the pretype holds before evaluating an expression a, then the property specified by the posttype after the evaluation. In our case, this says that, if some variable live before the evaluation is not in *live*, then some variable live after is not in $live'$, or, contrapositively (in the direction of the analysis), if all variables live after the evaluation are in $live'$, then all variables live before are in *live*. The typing rules state the constraints of the analysis. For live variables, they appear in Figure 4.4.

The rule for variables reflects the fact that a use of a variable makes it live (again in the direction of the analysis, i.e., backwards; this is also the direction for the comments about all other rules below). As a result the weakest pretype of an expression is obtained by adding its free variables to the posttype .

There are two rules for assignment, corresponding to the cases where the assigned variable x is in the posttype and where it is not. In the first case, since x is possibly live at the end, the variables in the expression a assigned to it should be included in the pretype. However, x itself should first be removed as the assignment kills it (it will reappear in the pretype, if it is among the variables in the expression a). In the second case, since x is necessarily dead at the end, there is no point in making the variables in a possibly live at the beginning.

The rule $\mathrm{comp}_{\mathrm{lv}}$ for composition should be self-explanatory. To type an if-statement, both of the branches have to have the same type (with the $\mathrm{conseq}_{\mathrm{lv}}$ rule, the pretypes may be strengthened to agree). Additionally, variables used in the

[1] This is the strong version of liveness. In the alternative weaker version, any use of a variable makes it live.

$$\frac{}{x : live \cup \{x\} \longrightarrow live}\ \mathrm{var_{lv}} \qquad \frac{}{n : live \longrightarrow live}\ \mathrm{num_{lv}} \qquad \frac{a_0 : live \longrightarrow live'' \quad a_1 : live'' \longrightarrow live'}{a_0 + a_1 : live \longrightarrow live'}\ +_{\mathrm{lv}}$$

$$\frac{a_0 : live \longrightarrow live'' \quad a_1 : live'' \longrightarrow live'}{a_0 = a_1 : live \longrightarrow live'}\ =_{\mathrm{lv}}$$

$$\frac{x \in live' \quad a : live \longrightarrow live' \setminus \{x\}}{x := a : live \longrightarrow live'}\ :=_{1\mathrm{lv}} \qquad \frac{x \notin live}{x := a : live \longrightarrow live}\ :=_{2\mathrm{lv}}$$

$$\frac{}{\mathsf{skip} : live \longrightarrow live}\ \mathrm{skip_{lv}} \qquad \frac{s_0 : live \longrightarrow live'' \quad s_1 : live'' \longrightarrow live'}{s_0; s_1 : live \longrightarrow live'}\ \mathrm{comp_{lv}}$$

$$\frac{b : live \longrightarrow live'' \quad s_t : live'' \longrightarrow live' \quad s_f : live'' \longrightarrow live'}{\mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f : live \longrightarrow live'}\ \mathrm{if_{lv}} \qquad \frac{b : live \longrightarrow live' \quad s_t : live' \longrightarrow live}{\mathsf{while}\ b\ \mathsf{do}\ s_t : live \longrightarrow live'}\ \mathrm{while_{lv}}$$

$$\frac{live \leq live_0 \quad s : live_0 \longrightarrow live'_0 \quad live'_0 \leq live'}{s : live \longrightarrow live'}\ \mathrm{conseq_{lv}}$$

Figure 4.4: Type system for live variables analysis

guard add to the pretype of the if-statement.

The rule $\mathrm{while_{lv}}$ requires an invariant-type for the beginning of the loop body/end of the guard to type a loop. The analysis computes it from a given posttype as the greatest fixpoint of a function monotone with respect to $\leq$. The type system accepts any fixpoint. The $\mathrm{conseq_{lv}}$ rule can be used to strengthen the given posttype to any suitable such type. There is also the invariant-type for the end of the loop body/beginning of the guard, obtained by adding the variables in the guard. There is an obvious similarity between the loop invariants here and those in Hoare logic.

The reason why the $\mathrm{while_{lv}}$ rule has the presented form can be made more clear on this example: $\mathsf{while}\ u < v\ \mathsf{do}\ (x := y; u := u + 1; y := z)$. If as a posttype of the loop we have the variable x (i.e., x is the only variable whose value we are interested in at the end), then in the naive approach (without strengthening it to the invariant for the beginning of the loop body) it would appear that the assignment to y is not necessary (while it clearly is needed, since the second time the loop is entered, its value has changed, and the changed value is assigned to x). Also, the fact that the assignment to u is not considered necessarily dead is because the invariant for the end of the loop body has the free variables of the guard already included.

The $\mathrm{conseq_{lv}}$ rule is a subsumption rule, but its role is completely analogous to that of the consequence rule in Hoare logic (except that checking subtyping is trivial whereas checking logical consequence needs a logic theorem prover, if no hints about the proof are supplied).

A big difference of the type system from the analysis algorithm is that while the algorithm computes the weakest preproperty for a given postproperty, the type system approves any valid pretype-posttype pair. Again, stronger pretypes are easy to get from the weakest one with $\mathrm{conseq_{lv}}$. The analysis algorithm can in fact be seen as an algorithm for principal type inference: given a statement s and a posttype $live'$, one attempts to construct a type derivation. Constructing the tightest one

$$\frac{x \in live' \quad a : live \longrightarrow live' \setminus \{x\}}{x := a : live \longrightarrow live' \hookrightarrow x := a} :=_{1}{}^{\text{opt}}_{\text{lv}} \qquad \frac{x \notin live}{x := a : live \longrightarrow live \hookrightarrow \mathsf{skip}} :=_{2}{}^{\text{opt}}_{\text{lv}}$$

$$\frac{}{\mathsf{skip} : live \longrightarrow live \hookrightarrow \mathsf{skip}} \text{skip}^{\text{opt}}_{\text{lv}} \qquad \frac{s_0 : live \longrightarrow live'' \hookrightarrow s_0' \quad s_1 : live'' \longrightarrow live' \hookrightarrow s_1'}{s_0; s_1 : live \longrightarrow live' \hookrightarrow s_0'; s_1'} \text{comp}^{\text{opt}}_{\text{lv}}$$

$$\frac{b : live \longrightarrow live'' \quad s_t : live'' \longrightarrow live' \hookrightarrow s_t' \quad s_f : live'' \longrightarrow live' \hookrightarrow s_f'}{\mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f : live \longrightarrow live' \hookrightarrow \mathsf{if}\ b\ \mathsf{then}\ s_t'\ \mathsf{else}\ s_f'} \text{if}^{\text{opt}}_{\text{lv}}$$

$$\frac{b : live \longrightarrow live' \quad s_t : live' \longrightarrow live \hookrightarrow s_t'}{\mathsf{while}\ b\ \mathsf{do}\ s_t : live \longrightarrow live' \hookrightarrow \mathsf{while}\ b\ \mathsf{do}\ s_t'} \text{while}^{\text{opt}}_{\text{lv}}$$

$$\frac{live \leq live_0 \quad s : live_0 \longrightarrow live_0' \hookrightarrow s' \quad live_0' \leq live'}{s : live \longrightarrow live' \hookrightarrow s'} \text{conseq}^{\text{opt}}_{\text{lv}}$$

Figure 4.5: Type system for dead code elimination

takes calculation of greatest fixpoints with respect to $\leq$ to obtain the invariant-types of the loops and as a result one learns the weakest pretype *live*. This type declares only these variables to be possibly live initially that really have some chance of being live. In type systems jargon, it makes sense to call *live* the principal type of s with respect to $live'$.

Soundness of live variable analysis with respect to the natural semantics can be conveniently formulated "relationally". Let $\sigma \sim_{live} \sigma'$ denote that two states σ and σ' agree on all variables in a set $live \subseteq \mathbf{Var}$, i.e., $\bigwedge_{x \in live} \sigma(x) = \sigma'(x)$. Then soundness states that any program is simulated by itself with respect to $\sim$. We look at this in more detail in the next section, where we discuss the optimization based on live variable analysis, namely dead code elimination.

4.2.2 Type system for dead code elimination

Dead code elimination removes from a statement the assignments that cannot affect the final values of the variables that are live at the end.

This optimization can be explained in an extended version of the live variables type system. Apart for assigning types to statements, it also defines their corresponding optimized forms. A typing judgement has the form $s : live \longrightarrow live' \hookrightarrow s'$, where s' is a statement; it says that s' is the optimized form of s. The rules of this extended type system are given in Figure 4.5. Arithmetic and boolean expressions are not optimized, so we do not repeat their rules.

The only rule where an actual optimization takes place is $:=_{2}{}^{\text{opt}}_{\text{lv}}$: if we know from the typing of an assignment that the assigned variable is necessarily dead after, then its value cannot affect any live variables. Thus we can replace the assignment with skip. We could add even stronger optimizations, for example removing a skip from a sequence or replacing an if-statement with skip, if both branches optimize to skip, but this can be seen as a separate optimization and we do not integrate it here.

An example of a derivation of a dead code elimination can be seen in Figure 4.6. In this example, we are interested in the program slice concerned with variable x.

$$\cfrac{\cfrac{\cfrac{\overline{x * 2 : \{x, y\} \longrightarrow \{y\}}}{x := x * 2 : \{x, y\} \longrightarrow \{x, y\} \hookrightarrow x := x * 2} \quad \cfrac{}{z := z + 1 : \{x, y\} \longrightarrow \{x, y\} \hookrightarrow \mathsf{skip}}}{x := x * 2; z := z + 1 : \{x, y\} \longrightarrow \{x, y\} \hookrightarrow x := x * 2; \mathsf{skip}}}{\cfrac{\mathsf{while}\ x < y\ \mathsf{do}\ (x := x * 2; z := z + 1) : \{x, y\} \longrightarrow \{x, y\} \hookrightarrow \mathsf{while}\ x < y\ \mathsf{do}\ (x := x * 2; \mathsf{skip})}{\mathsf{while}\ x < y\ \mathsf{do}\ (x := x * 2; z := z + 1) : \{x, y\} \longrightarrow \{x\} \hookrightarrow \mathsf{while}\ x < y\ \mathsf{do}\ (x := x * 2; \mathsf{skip})}}$$

Figure 4.6: An analysis and transformation of an example program

Thus the only variable in the posttype is x, and the code not affecting its final value is considered dead and thus removed.

The statement of soundness of dead code elimination is similar to that for the underlying analysis. Soundness says that the original and optimized form of a program simulate each other with respect to $\sim$.

Theorem 15 (Soundness of dead code elimination)
(o) If $live \leq live'$ *and* $\sigma \sim_{live} \sigma_*$, *then* $\sigma \sim_{live'} \sigma_*$.
(i) If $a : live \longrightarrow live'$ *and* $\sigma \sim_{live} \sigma_*$, *then* $[\![a]\!]\sigma = [\![a]\!]\sigma_*$ *and* $\sigma \sim_{live'} \sigma_*$.
(ii) If $b : live \longrightarrow live'$ *and* $\sigma \sim_{live} \sigma_*$, *then* $[\![b]\!]\sigma = [\![b]\!]\sigma_*$ *and* $\sigma \sim_{live'} \sigma_*$.
(iii) If $s : live \longrightarrow live' \hookrightarrow s'$ *and* $\sigma \sim_{live} \sigma_*$, *then*
— $\sigma \succ s \rightarrow \sigma'$ *implies the existence of* σ'_* *such that* $\sigma' \sim_{live'} \sigma'_*$ *and* $\sigma_* \succ s' \rightarrow \sigma'_*$,
— $\sigma_* \succ s' \rightarrow \sigma'_*$ *implies the existence of* σ' *such that* $\sigma' \sim_{live'} \sigma'_*$ *and* $\sigma \succ s \rightarrow \sigma'$.

Although our language is deterministic, nontermination is possible, therefore both directions (preservation and reflection of evaluations) are necessary to establish equitermination. Reflection in particular establishes that the optimized program cannot terminate more often than the original form.

Proof. (o) holds trivially, since $live \supseteq live'$.
We can see that (i) holds by inspecting the typing rules for a and seeing that $live$ includes all free variables of a, and $live \supseteq live'$. Thus all variables of a are equal in σ and σ_*, and consequently $[\![a]\!]\sigma = [\![a]\!]\sigma_*$. By (o), it also follows that $\sigma' \sim_{live'} \sigma'_*$. A similar argument can be made for (ii).

(iii) is proved by induction on the derivation of $s : live \longrightarrow live' \hookrightarrow s'$. We show the first part of (iii) (the other direction is analogous). We assume $\sigma \sim_{live} \sigma_*$ and $\sigma \succ s \rightarrow \sigma'$ and have the following cases:

- The type derivation is of the form

$$\frac{x \in live' \quad a : live \longrightarrow live' \setminus \{x\}}{x := a : live \longrightarrow live' \hookrightarrow x := a} :=_{1_{\mathrm{lv}}}^{\mathrm{opt}}$$

The given semantic derivation must be of the form

$$\frac{}{\sigma \succ x := a \rightarrow \sigma[x \mapsto [\![a]\!]\sigma]}$$

so $\sigma' = \sigma[x \mapsto [\![a]\!]\sigma]$. We have the semantic judgement $\sigma_* \succ x := a \rightarrow \sigma'_*$, where $\sigma'_* = \sigma_*[x \mapsto [\![a]\!]\sigma]$. From (i), it follows that $[\![a]\!]\sigma = [\![a]\!]\sigma_*$ and $\sigma' \sim_{live' \setminus \{x\}} \sigma'_*$. Since $[\![a]\!]\sigma = [\![a]\!]\sigma_*$, it means that $\sigma'(x) = \sigma'_*(x)$, and therefore we get $\sigma' \sim_{live'} \sigma'_*$

- The type derivation is of the form

$$\frac{x \notin live}{x := a : live \longrightarrow live \hookrightarrow \mathsf{skip}} :=2^{\text{opt}}_{\text{lv}}$$

The given semantic derivation must be of the form

$$\frac{}{\sigma \succ x := a \rightarrow \sigma[x \mapsto [\![a]\!]\sigma]}$$

so $\sigma' = \sigma[x \mapsto [\![a]\!]\sigma]$. We have the semantic judgement $\sigma_* \succ \mathsf{skip} \rightarrow \sigma_*$. Since $x \notin live$, from $\sigma \sim_{live} \sigma_*$ we may conclude that $\sigma' \sim_{live} \sigma'_*$.

- The type derivation is of the form

$$\frac{\begin{array}{c}\vdots \\ s_0 : live \longrightarrow live'' \hookrightarrow s'_0\end{array} \quad \begin{array}{c}\vdots \\ s_1 : live'' \longrightarrow live' \hookrightarrow s'_1\end{array}}{s_0; s_1 : live \longrightarrow live' \hookrightarrow s'_0; s'_1} \text{comp}^{\text{opt}}_{\text{lv}}$$

The given semantic judgement must be of the form

$$\frac{\begin{array}{c}\vdots \\ \sigma \succ s_0 \rightarrow \sigma''\end{array} \quad \begin{array}{c}\vdots \\ \sigma'' \succ s_1 \rightarrow \sigma'\end{array}}{\sigma \succ s_0; s_1 \rightarrow \sigma'}$$

For some σ''_*, σ'_* we have the semantic judgement

$$\frac{\begin{array}{c}\vdots\ IH \\ \sigma_* \succ s'_0 \rightarrow \sigma''_*\end{array} \quad \begin{array}{c}\vdots\ IH \\ \sigma''_* \succ s'_1 \rightarrow \sigma'_*\end{array}}{\sigma_* \succ s'_0; s'_1 \rightarrow \sigma'_*}$$

From the first induction hypothesis, we get that $\sigma'' \sim_{live''} \sigma''_*$, which allows us to invoke the second the second induction hypothesis from which we get that $\sigma' \sim_{live'} \sigma'_*$.

- The type derivation is of the form

$$\frac{b : live \longrightarrow live'' \quad s_t : live'' \longrightarrow live' \hookrightarrow s'_t \quad s_f : live'' \longrightarrow live' \hookrightarrow s'_f}{\mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f : live \longrightarrow live' \hookrightarrow \mathsf{if}\ b\ \mathsf{then}\ s'_t\ \mathsf{else}\ s'_f} \text{if}^{\text{opt}}_{\text{lv}}$$

We have that either $\sigma \models b$ or $\sigma \not\models b$. In the first case, the given semantic derivation is of the form

$$\frac{\sigma \models b \quad \begin{array}{c}\vdots\\ \sigma \succ s_t \rightarrow \sigma'\end{array}}{\sigma \succ \textsf{if } b \textsf{ then } s_t \textsf{ else } s_f \rightarrow \sigma'}$$

From (ii), it follows that $\sigma(b) = \sigma_*(b)$ and therefore if $\sigma \models b$ then $\sigma_* \models b$. From (ii) it also follows that $\sigma \sim_{live''} \sigma_*$. We thus have the derivation

$$\frac{\sigma_* \models b \quad \begin{array}{c}\vdots \; IH\\ \sigma_* \succ s'_t \rightarrow \sigma'_*\end{array}}{\sigma_* \succ \textsf{if } b \textsf{ then } s'_t \textsf{ else } s'_f \rightarrow \sigma'_*}$$

for some σ'_*.

By the induction hypothesis, we also get that $\sigma' \sim_{live'} \sigma'_*$

Similar reasoning holds for $\sigma \not\models b$.

- The type derivation is of the form

$$\frac{\begin{array}{c}\vdots\\ b : live \longrightarrow live'\end{array} \quad \begin{array}{c}\vdots\\ s_t : live' \longrightarrow live \hookrightarrow s'_t\end{array}}{\textsf{while } b \textsf{ do } s_t : live \longrightarrow live' \hookrightarrow \textsf{while } b \textsf{ do } s'_t} \; \text{while}^{\text{opt}}_{\text{lv}}$$

We also invoke structural induction on the given semantic derivation of $\sigma \succ \textsf{while } b \textsf{ do } s_t \rightarrow \sigma'$. We have that either $\sigma \models b$ or $\sigma \not\models b$. In the first case, the given semantic derivation is of the form

$$\frac{\sigma \models b \quad \begin{array}{c}\vdots\\ \sigma \succ s_t \rightarrow \sigma''\end{array} \quad \begin{array}{c}\vdots\\ \sigma'' \succ \textsf{while } b \textsf{ do } s_t \rightarrow \sigma'\end{array}}{\sigma \succ \textsf{while } b \textsf{ do } s_t \rightarrow \sigma'}$$

From (ii), it follows that $\sigma(b) = \sigma_*(b)$ and therefore if $\sigma \models b$ then $\sigma_* \models b$. From (ii) it also follows that $\sigma \sim_{live''} \sigma_*$. We thus have the derivation

$$\frac{\sigma_* \models b \quad \begin{array}{c}\vdots \; \text{outer } IH\\ \sigma_* \succ s'_t \rightarrow \sigma''_*\end{array} \quad \begin{array}{c}\vdots \; \text{inner } IH\\ \sigma''_* \succ \textsf{while } b \textsf{ do } s'_t \rightarrow \sigma'_*\end{array}}{\sigma_* \succ \textsf{while } b \textsf{ do } s'_t \rightarrow \sigma'_*}$$

for some σ'_* and σ''_*.

By the outer induction hypothesis, we get that $\sigma'' \sim_{live} \sigma''_*$. From the inner induction hypothesis, we get $\sigma' \sim_{live'} \sigma'_*$

If $\sigma \not\models b$, then the semantic derivation must be

$$\frac{\sigma \not\models b}{\sigma \succ \textsf{while } b \textsf{ do } s_t \to \sigma}$$

Again by (ii), we have that $\sigma_* \not\models b$. We have the derivation

$$\frac{\sigma \not\models b}{\sigma_* \succ \textsf{while } b \textsf{ do } s'_t \to \sigma_*}$$

The fact that $\sigma' \sim_{live'} \sigma'_*$ follows trivially.

- The type derivation is of the form

$$\frac{live \leq live_0 \quad s : live_0 \longrightarrow live'_0 \hookrightarrow s' \quad live'_0 \leq live}{s : live \longrightarrow live' \hookrightarrow s'} \text{conseq}^{\text{opt}}_{\text{lv}}$$

  We also have the given semantic judgement $\sigma \succ s \to \sigma'$. Since $live \supseteq live_0$, from $\sigma \sim_{live} \sigma_*$ we obtain that $\sigma \sim_{live_0} \sigma_*$. By the induction hypothesis, there must be a state σ'_* such that $\sigma_* \succ s' \to \sigma'_*$ and $\sigma' \sim_{live'_0} \sigma'_*$. Since $live'_0 \supseteq live'$, we have that $\sigma' \sim_{live'} \sigma'_*$.

□

We now arrive at our main motivation for the type-systematic setup, namely proof transformation. It is easy to see that Theorem 15 has a counterpart for the Hoare logic. Essentially, it says that optimization preserves and reflects Hoare triple derivability (in fact even actual derivations).

Let $P|_{live}$ abbreviate the formula $\exists[v(x) \mid x \notin live](P[v(x)/x \mid x \notin live])$, where v is some assignment of unique logic variable names to program variables (so that, informally, $P|_{live}$ is obtained from P by quantifying out all program variables not in *live*). For example for the assertion $P =_{\text{df}} x = 2 \wedge y = 7$ and type $live = \{x\}$, $P|_{live}$ is $\exists y' \, (x = 2 \wedge y' = 7)$.

Theorem 16
(o) If $live \leq live'$, then $P|_{live} \models P|_{live'}$.
(i) If $a : live \longrightarrow live'$, then $(P[a/w])|_{live} \models (P|_{live'})[a/w]$ (where w is a logic variable).
(ii) If $b : live \longrightarrow live'$, then $(P[b/w])|_{live} \models (P|_{live'})[b/w]$ (where w is a logic variable). As a consequence, $(P|_{live}) \models b \Rightarrow ((b \wedge P)|_{live'})$ and $(P|_{live}) \models \neg b \Rightarrow ((\neg b \wedge P)|_{live'})$.
(iii) If $s : live \longrightarrow live' \hookrightarrow s'$ and $\{P\}\, s\, \{Q\}$, then also $\{P|_{live}\}\, s'\, \{Q|_{live'}\}$.

The theorem can be concluded from Theorem 15 using the soundness and completeness of the Hoare logic. But it is also provable constructively, without any indirection via semantics, by induction on the structure of the type derivation. The

proof of (iii) gives a transformation of a given Hoare triple derivation into a derivation for the modified triple.

Proof. (Constructive proof)

(o) The assumption $\mathit{live} \leq \mathit{live}'$ means $\mathit{live} \supseteq \mathit{live}'$. Therefore, $P|_{\mathit{live}} \models P|_{\mathit{live}'}$ follows by existential introduction: for any variable $x \in \mathit{live} \setminus \mathit{live}'$, a suitable constructed witness for $v(x)$ is x.

(i) We construct a derivation of $(P[a/w])|_{\mathit{live}} \models (P|_{\mathit{live}'})[a/w]$ by induction on the derivation of $a : \mathit{live} \longrightarrow \mathit{live}'$, which gives the following cases.

- The type derivation is

$$\frac{}{x : \mathit{live} \cup \{x\} \longrightarrow \mathit{live}}\ \mathrm{var_{lv}}$$

  The required entailment $(P[x/w])|_{\mathit{live}\cup\{x\}} \models (P|_{\mathit{live}})[x/w]$ follows from $(P[x/w])|_{\mathit{live}\cup\{x\}} \equiv P|_{\mathit{live}\cup\{x\}}[x/w]$, which holds trivially, and $P|_{\mathit{live}\cup\{x\}}[x/w] \models P|_{\mathit{live}}[x/w]$, which results from $P|_{\mathit{live}\cup\{x\}} \models P|_{\mathit{live}}$ from existential introduction (taking x as the constructed witness for $v(x)$), if $x \notin \mathit{live}$.

- The type derivation is

$$\frac{}{n : \mathit{live} \longrightarrow \mathit{live}}\ \mathrm{num_{lv}}$$

  The required entailment holds trivially: we have $(P[n/w])|_{\mathit{live}} \equiv (P|_{\mathit{live}})[n/w]$.

- The type derivation is of the form

$$\frac{\begin{matrix}\vdots & \vdots \\ a_0 : \mathit{live} \longrightarrow \mathit{live}'' & a_1 : \mathit{live}'' \longrightarrow \mathit{live}'\end{matrix}}{a_0 + a_1 : \mathit{live} \longrightarrow \mathit{live}'}\ +_{\mathrm{lv}}$$

  The required entailment $(P[a_0+a_1/w])|_{\mathit{live}} \models (P|_{\mathit{live}'})[a_0+a_1/w]$ follows from the following chain of entailments:

$$\begin{array}{lll} (P[a_0 + a_1/w])|_{\mathit{live}} = & & \\ \quad = & (P[w_0 + w_1/w][a_1/w_1][a_0/w_0])|_{\mathit{live}} & w_0,\ w_1 \text{ are not free in } P \\ \quad \models & ((P[w_0 + w_1/w][a_1/w_1])|_{\mathit{live}''})[a_0/w_0] & \text{by IH} \\ \quad \models & ((P[w_0 + w_1/w])|_{\mathit{live}'})[a_1/w_1][a_0/w_0] & \text{by IH} \\ \quad = & (P|_{\mathit{live}'})[w_0 + w_1/w][a_1/w_1][a_0/w_0] & \text{trivially} \\ \quad = & (P|_{\mathit{live}'})[a_0 + a_1/w] & w_0,\ w_1 \text{ are not free in } P \end{array}$$

  where w_0 and w_1 are logic variables distinct from the free logic variables of P.

(ii) is proved similarly to (i).

(iii) We construct a derivation of $\{P|_{\mathit{live}}\}\, s\, \{Q|_{\mathit{live}'}\}$ by induction on the derivation of $s : \mathit{live} \longrightarrow \mathit{live}'$ and inspection of the derivation of $\{P\}\, s\, \{Q\}$. We have the following cases.

- The type derivation is of the form

$$\dfrac{x \in \mathit{live}' \quad \dfrac{\vdots}{a : \mathit{live} \longrightarrow \mathit{live}' \setminus \{x\}}}{x := a : \mathit{live} \longrightarrow \mathit{live}' \hookrightarrow x := a} :=_{1\text{lv}}$$

  and the given Hoare derivation is

$$\overline{\{P[a/x]\}\, x := a\, \{P\}}$$

  We get this modified Hoare triple derivation:

$$\dfrac{(P[a/x])|_{\mathit{live}} \models (P|_{\mathit{live}'})[a/x] \quad \overline{\{(P|_{\mathit{live}'})[a/x]\}\, x := a\, \{P|_{\mathit{live}'}\}}}{\{(P[a/x])|_{\mathit{live}}\}\, x := a\, \{P|_{\mathit{live}'}\}}$$

  The entailment $(P[a/x])|_{\mathit{live}} \models (P|_{\mathit{live}'})[a/x]$ is the consequence of the following chain of entailments:

$$\begin{array}{rcll}
(P[a/x])|_{\mathit{live}} & = & (P[w/x][a/w])|_{\mathit{live}} & w \text{ is not free in } P \\
 & \models & ((P[w/x])|_{\mathit{live}' \setminus \{x\}})[a/w] & \text{by (i)} \\
 & \Leftrightarrow & ((P[w/x])|_{\mathit{live}'})[a/w] & x \text{ does not occur in } P[w/x] \\
 & = & (P|_{\mathit{live}'})[w/x][a/w] & x \in \mathit{live}' \\
 & = & (P|_{\mathit{live}'})[a/x] & w \text{ is not free in } P
\end{array}$$

  where w is a logic variable distinct from the free logic variables of P.

- The type derivation is of the form

$$\dfrac{x \notin \mathit{live}}{x := a : \mathit{live} \longrightarrow \mathit{live} \hookrightarrow \mathsf{skip}} :=_{2\text{lv}}$$

  and the given Hoare triple derivation is

$$\overline{\{P[a/x]\}\, x := a\, \{P\}}$$

  For the optimized program skip, we have the Hoare triple derivation

$$\dfrac{P[a/x]|_{\mathit{live}} \models P|_{\mathit{live}} \quad \overline{\{P|_{\mathit{live}}\}\, \mathsf{skip}\, \{P|_{\mathit{live}}\}}}{\{P[a/x]|_{\mathit{live}}\}\, \mathsf{skip}\, \{P|_{\mathit{live}}\}}$$

  The entailment $(P[a/x])|_{\mathit{live}} \models (P|_{\mathit{live}})[a/x]$ is a consequence of $(P[a/x])|_{\mathit{live}} \models P|_{\mathit{live}}$, which holds by $x \notin \mathit{live}$ and existential elimination and introduction (for the constructed witness of $v(x)$ on the right one must take $a[w/x]$ where w is the assumed witness of $v(x)$ on the left), and $P|_{\mathit{live}} \equiv (P|_{\mathit{live}})[a/x]$, which also holds since $x \notin \mathit{live}$, as $P|_{\mathit{live}}$ has therefore no occurrences of x.

- The type derivation is of the form

$$\dfrac{\begin{array}{c}\vdots\\ s_0 : \mathit{live} \longrightarrow \mathit{live}'' \hookrightarrow s_0'\end{array} \quad \begin{array}{c}\vdots\\ s_1 : \mathit{live}'' \longrightarrow \mathit{live}' \hookrightarrow s_1'\end{array}}{s_0; s_1 : \mathit{live} \longrightarrow \mathit{live}' \hookrightarrow s_0'; s_1'}\ \mathrm{comp}_{\mathrm{lv}}$$

and the given Hoare derivation is of the form

$$\dfrac{\begin{array}{c}\vdots\\ \{P\}\, s_0\, \{R\}\end{array} \quad \begin{array}{c}\vdots\\ \{R\}\, s_1\, \{Q\}\end{array}}{\{P\}\, s_0; s_1\, \{Q\}}\ .$$

We get the Hoare derivation

$$\dfrac{\begin{array}{c}\vdots\ \mathit{IH}\\ \{P|_{\mathit{live}}\}\, s_0'\, \{R|_{\mathit{live}''}\}\end{array} \quad \begin{array}{c}\vdots\ \mathit{IH}\\ \{R|_{\mathit{live}''}\}\, s_0'\, \{Q|_{\mathit{live}'}\}\end{array}}{\{P|_{\mathit{live}}\}\, s_0'; s_1'\, \{Q|_{\mathit{live}'}\}}$$

- The type derivation is of the form

$$\dfrac{\begin{array}{c}\vdots\\ b : \mathit{live} \longrightarrow \mathit{live}''\end{array} \quad \begin{array}{c}\vdots\\ s_t : \mathit{live}'' \longrightarrow \mathit{live}' \hookrightarrow s_t'\end{array} \quad \begin{array}{c}\vdots\\ s_f : \mathit{live}'' \longrightarrow \mathit{live}' \hookrightarrow s_f'\end{array}}{\mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f : \mathit{live} \longrightarrow \mathit{live}' \hookrightarrow \mathsf{if}\ b\ \mathsf{then}\ s_t'\ \mathsf{else}\ s_f'}\ \mathrm{if}_{\mathrm{lv}}$$

and the given Hoare triple derivation is of the form

$$\dfrac{\begin{array}{c}\vdots\\ \{b \wedge P\}\, s_t\, \{Q\}\end{array} \quad \begin{array}{c}\vdots\\ \{\neg b \wedge P\}\, s_f\, \{Q\}\end{array}}{\{P\}\ \mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f\, \{Q\}}\ .$$

We have the Hoare triple derivation

$$\dfrac{\dfrac{\begin{array}{l}P|_{\mathit{live}} \models\\ b \Rightarrow (b \wedge P)|_{\mathit{live}''}\end{array} \quad \begin{array}{c}\vdots\ \mathit{IH}\\ \{(b \wedge P)|_{\mathit{live}''}\}\, s_t'\, \{Q|_{\mathit{live}'}\}\end{array}}{\{b \wedge (P|_{\mathit{live}})\}\, s_t'\, \{Q|_{\mathit{live}'}\}} \quad \dfrac{\begin{array}{l}P|_{\mathit{live}} \models\\ \neg b \Rightarrow (\neg b \wedge P)|_{\mathit{live}''}\end{array} \quad \begin{array}{c}\vdots\ \mathit{IH}\\ \{(\neg b \wedge P)|_{\mathit{live}''}\}\, s_f'\, \{Q|_{\mathit{live}'}\}\end{array}}{\{\neg b \wedge (P|_{\mathit{live}})\}\, s_f'\, \{Q|_{\mathit{live}'}\}}}{\{P|_{\mathit{live}}\}\ \mathsf{if}\ b\ \mathsf{then}\ s_t'\ \mathsf{else}\ s_f'\, \{Q|_{\mathit{live}'}\}}$$

The two entailments hold by (ii).

- The type derivation is of the form

$$\frac{\begin{array}{c}\vdots\\ b : \mathit{live} \longrightarrow \mathit{live}'\end{array} \quad \begin{array}{c}\vdots\\ s_t : \mathit{live}' \longrightarrow \mathit{live} \hookrightarrow s_t'\end{array}}{\mathsf{while}\ b\ \mathsf{do}\ s_t : \mathit{live} \longrightarrow \mathit{live}' \hookrightarrow \mathsf{while}\ b\ \mathsf{do}\ s_t'}\ \text{while}_{\text{lv}}$$

  and the given Hoare triple derivation of the form

$$\frac{\begin{array}{c}\vdots\\ \{b \wedge P\}\, s_t\, \{P\}\end{array}}{\{P\}\, \mathsf{while}\ b\ \mathsf{do}\ s_t\, \{\neg b \wedge P\}}.$$

  We have the Hoare triple derivation

$$\frac{\dfrac{\dfrac{(P|_{\mathit{live}}) \models b \Rightarrow (b \wedge P)|_{\mathit{live}'} \quad \begin{array}{c}\vdots\ \mathit{IH}\\ \{(b \wedge P)|_{\mathit{live}'}\}\, s_t'\, \{P|_{\mathit{live}}\}\end{array}}{\{b \wedge (P|_{\mathit{live}})\}\, s_t\, \{P|_{\mathit{live}}\}}}{\{P|_{\mathit{live}}\}\, \mathsf{while}\ b\ \mathsf{do}\ s_t'\, \{\neg b \wedge (P|_{\mathit{live}})\}} \quad (P|_{\mathit{live}}) \models \neg b \Rightarrow (\neg b \wedge P)|_{\mathit{live}'}}{\{P|_{\mathit{live}}\}\, \mathsf{while}\ b\ \mathsf{do}\ s_t'\, \{(\neg b \wedge P)|_{\mathit{live}'}\}}$$

  The two entailments hold by (ii).

- The type derivation is of the form

$$\frac{\mathit{live} \leq \mathit{live}_0 \quad \begin{array}{c}\vdots\\ s : \mathit{live}_0 \longrightarrow \mathit{live}_0' \hookrightarrow s'\end{array} \quad \mathit{live}_0' \leq \mathit{live}'}{s : \mathit{live} \longrightarrow \mathit{live}' \hookrightarrow s'}\ \text{conseq}_{\text{lv}}$$

  and the given Hoare triple derivation is of the form

$$\frac{P \models P' \quad \begin{array}{c}\vdots\\ \{P'\}\, s\, \{Q'\}\end{array} \quad Q' \models Q}{\{P\}\, s\, \{Q\}}.$$

  We have the following Hoare triple derivation:

$$\frac{P|_{\mathit{live}} \models P'|_{\mathit{live}_0} \quad \begin{array}{c}\vdots\ \mathit{IH}\\ \{P'|_{\mathit{live}_0}\}\, s'\, \{Q'|_{\mathit{live}_0'}\}\end{array} \quad Q'|_{\mathit{live}_0'} \models Q|_{\mathit{live}'}}{\{P|_{\mathit{live}}\}\, s'\, \{Q|_{\mathit{live}'}\}}$$

  The entailment $P|_{\mathit{live}} \models P'|_{\mathit{live}_0}$ follows from $P|_{\mathit{live}} \models P'|_{\mathit{live}}$, which holds by $P \models P'$, and $P'|_{\mathit{live}} \models P'|_{\mathit{live}_0}$, which holds by (o). Similarly for $Q'|_{\mathit{live}_0'} \models Q|_{\mathit{live}'}$.

$$\{x = 2^{z+1} \wedge z+1 \leq ceil(\log y)\}\ z := z+1\ \{x = 2^z \wedge z \leq ceil(\log y)\}$$

$$\{x*2 = 2^{z+1} \wedge z+1 \leq ceil(\log y)\}\ x := x*2\ \{x = 2^{z+1} \wedge z+1 \leq ceil(\log y)\}$$

$$\{x*2 = 2^{z+1} \wedge z+1 \leq ceil(\log y)\}\ x := x*2; z := z+1\ \{x = 2^z \wedge z \leq ceil(\log y)\}$$

$$\{x < y \wedge x = 2^z \wedge z \leq ceil(\log y)\}\ x := x*2; z := z+1\ \{x = 2^z \wedge z \leq ceil(\log y)\}$$

$$\{x = 2^z \wedge z \leq ceil(\log y)\}\ \mathsf{while}\ x < y\ \mathsf{do}\ (x := x*2; z := z+1)\ \{x \not< y \wedge x = 2^z \wedge z \leq ceil(\log y)\}$$

$$\{x = 2 \wedge z = 1 \wedge y > 1\}\ \mathsf{while}\ x < y\ \mathsf{do}\ (x := x*2; z := z+1)\ \{x = 2^z \wedge z = ceil(\log y)\}$$

Figure 4.7: A proof of the example program

□

Theorem 16 gives us "proof optimization". Given a Hoare triple derivation for an original program, we get a modified Hoare triple and its derivation for its optimized form. In the example of program analysis and optimization in Figure 4.6 we saw that the program

$$\mathsf{while}\ x < y\ \mathsf{do}\ (x := x*2; z := z+1)$$

admits the type $\{x, y\} \longrightarrow \{x\}$ and that the corresponding optimized form is $\mathsf{while}\ x < y\ \mathsf{do}\ (x := x*2; \mathsf{skip})$ (further simplifiable to $\mathsf{while}\ x < y\ \mathsf{do}\ x := x*2$ by a trivial post-processing pass based on the equivalence $s; \mathsf{skip} = s$).

A Hoare logic derivation for the triple

$$\{x = 2 \wedge z = 1 \wedge y > 1\}$$
$$\mathsf{while}\ x < y\ \mathsf{do}\ (x := x*2; z := z+1)$$
$$\{x = 2^z \wedge z = ceil(\log y)\}$$

is given in Figure 4.7. (In order to save space, we have not spelled out the side conditions of inferences by the consequence rule.)

This Hoare triple derivation is mechanically transformable, following the given type derivation, into the derivation of the modified Hoare triple

$$\{\exists z'\,(x = 2 \wedge z' = 1 \wedge y > 1)\}$$
$$\mathsf{while}\ x < y\ \mathsf{do}\ x := x*2$$
$$\{\exists y', z'\,(x = 2^{z'} \wedge z' = ceil(\log y'))\}$$

given in Figure 4.8. (Note the inference by the consequence rule after the rule for skip: this involves a change of witness for z' as described in the proof of Theorem 16, a form of a shadow of the assignment $z := z+1$ in the original program.) The modified Hoare triple is equivalent to

$$\{x = 2 \wedge y > 1\}\mathsf{while}\ x < y\ \mathsf{do}\ x := x*2\{\exists z' > 0\,(x = 2^{z'})\}.$$

$$\frac{}{\{\exists z'\,(x = 2^{z'} \wedge z' \leq ceil(\log y))\}\ \mathsf{skip}\ \{\exists z'\,(x = 2^{z'} \wedge z' \leq ceil(\log y))\}}$$

$$\{\exists z'\,(x = 2^{z'+1} \wedge z' + 1 \leq ceil(\log y))\}\ \mathsf{skip}\ \{\exists z'\,(x = 2^{z'} \wedge z' \leq ceil(\log y))\}$$

$$\frac{}{\{\exists z'\,(x * 2 = 2^{z'+1} \wedge z' + 1 \leq ceil(\log y))\}\ x := x * 2\ \{\exists z'\,(x = 2^{z'+1} \wedge z' + 1 \leq ceil(\log y))\}}$$

$$\{\exists z'\,(x * 2 = 2^{z'+1} \wedge z' + 1 \leq ceil(\log y))\}\ x := x * 2; \mathsf{skip}\ \{\exists z'\,(x = 2^{z'} \wedge z' \leq ceil(\log y))\}$$

$$\{x < y \wedge \exists z'\,(x = 2^{z'} \wedge z' \leq ceil(\log y))\}\ x := x * 2; \mathsf{skip}\ \{\exists z'\,(x = 2^{z'} \wedge z' \leq ceil(\log y))\}$$

$$\{\exists z'\,(x = 2^{z'} \wedge z' \leq ceil(\log y))\}\ \mathsf{while}\ x < y\ \mathsf{do}\ (x := x * 2; \mathsf{skip})\ \{x \not< y \wedge \exists z'\,(x = 2^{z'} \wedge z' \leq ceil(\log y))\}$$

$$\{\exists z'\,(x = 2 \wedge z' = 1 \wedge y > 1)\}\ \mathsf{while}\ x < y\ \mathsf{do}\ (x := x * 2; \mathsf{skip})\ \{\exists z'\,(x = 2^{z'} \wedge z = ceil(\log y))\}$$

$$\{\exists z'\,(x = 2 \wedge z' = 1 \wedge y > 1)\}\ \mathsf{while}\ x < y\ \mathsf{do}\ (x := x * 2; \mathsf{skip})\ \{\exists y', z'\,(x = 2^{z'} \wedge z' = ceil(\log y'))\}$$

Figure 4.8: Transformed proof

4.3 Common subexpression elimination

4.3.1 Type system for available expressions analysis

We now look at another, more involved optimization called common subexpression elimination. The idea in common subexpression elimination is to avoid re-evaluation of non-trivial expressions. This is considerably more complicated and subtle than dead code elimination. The first phase in this optimization is the analysis of available expressions.

An (non-trivial arithmetic) expression is *available* at a program point, if every path to it (a) contains an evaluation of this expression (as a subexpression of an assigned expression or the guard of an if- or while-statement) and (b) does not contain a later modification (an assignment to a variable of the expression). The available expressions analysis finds, for each program point, which expressions *must* be available at that point. It is a *forward* analysis and starts from the set of expressions that one wishes to regard as available at the beginning of the program (typically, this would be $\emptyset$).

The types and subtyping of the type system for available expressions are $(D, \leq) =_{\mathrm{df}} (\mathcal{P}(\mathbf{AExp}^+), \supseteq)$. A state on a computation path has type $av \in D$, if all expressions in av are available in that state. A typing judgement for an arithmetic expression has the form $a : av \longrightarrow av'$ and means that, if all expressions in av are available before an evaluation of a, then all expressions in av' are available after the evaluation (for boolean expressions and statements, they are similar).Variables and numerals do not change the availability of expressions.

The typing rules appear in Figure 4.9. We use $mod(x)$ to denote the set of nontrivial arithmetic expressions containing x, i.e., $mod(x) =_{\mathrm{df}} \{a \mid x \in FV(a)\}$. The rule $+_{\mathrm{ae}}^{\mathrm{ts}}$ expresses that a compound expression makes itself and the subexpressions of its operands available. The rules for boolean expressions are similar. The rule $:=_{\mathrm{ae}}^{\mathrm{ts}}$ says that, after an assignment $x := a$, the arithmetic subexpressions of

$$\frac{}{x : av \longrightarrow av}\ \mathrm{var}^{\mathrm{ts}}_{\mathrm{ae}} \qquad \frac{}{n : av \longrightarrow av}\ \mathrm{num}^{\mathrm{ts}}_{\mathrm{ae}} \qquad \frac{a_0 : av \longrightarrow av'' \quad a_1 : av'' \longrightarrow av'}{a_0 + a_1 : av \longrightarrow av' \cup \{a_0 + a_1\}}\ +^{\mathrm{ts}}_{\mathrm{ae}}$$

$$\frac{a_0 : av \longrightarrow av'' \quad a_1 : av'' \longrightarrow av'}{a_0 = a_1 : av \longrightarrow av'}\ =^{\mathrm{ts}}_{\mathrm{ae}}$$

$$\frac{a : av \longrightarrow av'}{x := a : av \longrightarrow av' \backslash mod(x)}\ :=^{\mathrm{ts}}_{\mathrm{ae}}$$

$$\frac{}{\mathsf{skip} : av \longrightarrow av}\ \mathrm{skip}^{\mathrm{ts}}_{\mathrm{ae}} \qquad \frac{s_0 : av \longrightarrow av'' \quad s_1 : av'' \longrightarrow av'}{s_0; s_1 : av \longrightarrow av'}\ \mathrm{comp}^{\mathrm{ts}}_{\mathrm{ae}}$$

$$\frac{b : av \longrightarrow av'' \quad s_t : av'' \longrightarrow av' \quad s_f : av'' \longrightarrow av'}{\mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f : av \longrightarrow av'}\ \mathrm{if}^{\mathrm{ts}}_{\mathrm{ae}} \qquad \frac{b : av \longrightarrow av' \quad s_t : av' \longrightarrow av}{\mathsf{while}\ b\ \mathsf{do}\ s_t : av \longrightarrow av'}\ \mathrm{while}^{\mathrm{ts}}_{\mathrm{ae}}$$

$$\frac{av \leq av_0 \quad s : av_0 \longrightarrow av'_0 \quad av'_0 \leq av'}{s : av \longrightarrow av'}\ \mathrm{conseq}^{\mathrm{ts}}_{\mathrm{ae}}$$

Figure 4.9: Type system for available expressions analysis

expression a have been computed and are thus available. However, since x was assigned to, any precomputed value of an expression containing x is effectively killed. The skip and composition rules should be self-explanatory. The rule $\mathrm{if}^{\mathrm{ts}}_{\mathrm{ae}}$ says that if both branches of an if-statement have the same typing, we can give their posttype to the whole statement. But since the guard is always evaluated before either of the branch, the pretype of both branches is the posttype of the guard. The rule $\mathrm{while}^{\mathrm{ts}}_{\mathrm{ae}}$ requires an invariant-type for the beginning of the guard/end of the body of the loop, which will become the pretype of the loop itself. A given pretype for the loop can be weakened to this type using the $\mathrm{conseq}^{\mathrm{ts}}_{\mathrm{ae}}$ rule.

A type derivation of a program gives us two kinds of information. Firstly, based on the typing, we know where an expression first becomes available: where the expression is not available in its pretype. Secondly, it tells us where a pre-computed value can be used: where it is available in the pretype.

It would be possible to state and prove that the available expressions analysis is sound. We refrain from doing it here, as this requires an instrumentation of the standard natural semantics (the concept of state must be adjusted to record the last computed value of every non-trivial arithmetic expression, and the evaluation relation of the semantics must be adjusted accordingly). But we will state and prove the soundness of common subexpression elimination.

4.3.2 Type system for conditional partial anticipability analysis

The technique behind common subexpression elimination is to save computed values of expressions in new variables and to use these saved values instead of re-evaluating the expressions.

Although from the available expression analysis, we know where a particular expression becomes available and where a pre-computed value can be used, this information is not enough for the purpose of common subexpression elimination. The

reason is of course that a new variable to save the computation of the subexpression should only be introduced or updated when that subexpression is possibly used later on at a point where it is available. However, at the program point where an expression becomes available, we do not have that information from available expressions analysis. We would need to know what we call *conditionally partially anticipable* (cpant) expressions. We say that an expression is cpant at a program point, if there is a path from that program point that (a) contains an evaluation of it where the expression is available at the beginning of the evaluation, and (b) does not contain an earlier evaluation of it.

The need for the expression to be available before becoming anticipable in the reverse control-flow graph can be explained through the following example:

$$\underbrace{\mathsf{if}\ b\ \mathsf{then}\ \overbrace{x := 3}^{s_2}\ \mathsf{else}\ \overbrace{x := z + u * v}^{s_3}}_{s_1};\underbrace{y := u * v}_{s_4}.$$

The expression $u * v$ is not available after statement s_1, since it is not evaluated in both of its branches. So the expression can not be used for optimization at statement s_4 and therefore a new variable should not be introduced at statement s_3. The type system for cpant can give us this information; since the expression $u * v$ is not available before statement s_4, the statement, although using $u * v$, does not make it anticipable.

This analysis decides which expressions *may* be cpant at each program point. It relies on the results of the available expressions analysis and is a *backward* analysis. This analysis removes the need for establishing explicit "use-def" chains for expressions, i.e., associations of program points where an expression is evaluated ("defined") to program points where the computed value could be reused ("used"). Instead, it finds, for each program point, the expressions for which there can be a value reuse point (a future point where, for some reason, one could use the present value of an expression, or even a previously stored value of it, if it is presently available).

Since the cpant type system must use the typings from the available expressions type system, it is an extension. The types and the subtyping relation are $(D, \leq) =_{\mathrm{df}} (\{(av, cpant) \in \mathcal{P}(\mathbf{AExp}^+) \times \mathcal{P}(\mathbf{AExp}^+) \mid cpant \subseteq av\}, \supseteq \times \supseteq)$. A state (on a computation path) is in a type $(av, cpant) \in D$ if in that state all expressions in av are available and some cpant expression is not in *cpant* (remember that since cpant analysis is a backward one, the intuitive reading is contrapositive to the forward one). A typing judgement for an arithmetic expression (or a boolean expression, or a statement) is therefore of the form $a : av, cpant \longrightarrow av', cpant'$ and says that, if all expressions in av are available before an evaluation of a, then all expressions of av' are available after the evaluation, and, moreover, if all expressions av are available before an evaluation and all expressions cpant after the evaluation are in $cpant'$, then all expressions cpant before the evaluation are in *cpant*.

The typing rules are given in Figure 4.10. The key rule is $+^{\mathrm{ts}}_{\mathrm{le}}$. As was explained, an expression is made cpant at the beginning (i.e., included in the cpant

$$\frac{}{x : av, cpant \longrightarrow av, cpant}\ \text{var}^{\text{ts}}_{\text{le}} \qquad \frac{}{n : av, cpant \longrightarrow av, cpant}\ \text{num}^{\text{ts}}_{\text{le}}$$

$$\frac{a_0 : av, cpant \longrightarrow av'', cpant'' \quad a_1 : av'', cpant'' \longrightarrow av', cpant'}{a_0 + a_1 : av, cpant \cup (\{a_0 + a_1\} \cap av) \longrightarrow av' \cup \{a_0 + a_1\}, cpant'}\ +^{\text{ts}}_{\text{le}}$$

$$\frac{a_0 : av, cpant \longrightarrow av'', cpant'' \quad a_1 : av'', cpant'' \longrightarrow av', cpant'}{a_0 = a_1 : av, cpant \longrightarrow av', cpant'}\ =^{\text{ts}}_{\text{le}}$$

$$\frac{a : av, cpant \longrightarrow av', cpant'}{x := a : av, cpant \longrightarrow av' \setminus mod(x), cpant'}\ :=^{\text{ts}}_{\text{le}}$$

$$\frac{}{\mathsf{skip} : av, cpant \longrightarrow av, cpant}\ \text{skip}^{\text{ts}}_{\text{le}} \qquad \frac{s_0 : av, cpant \longrightarrow av'', cpant'' \quad s_1 : av'', cpant'' \longrightarrow av', cpant'}{s_0; s_1 : av, cpant \longrightarrow av', cpant'}\ \text{comp}^{\text{ts}}_{\text{le}}$$

$$\frac{b : av, cpant \longrightarrow av'', cpant'' \quad s_t : av'', cpant'' \longrightarrow av', cpant' \quad s_f : av'', cpant'' \longrightarrow av', cpant'}{\mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f : av, cpant \longrightarrow av', cpant'}\ \text{if}^{\text{ts}}_{\text{le}}$$

$$\frac{b : av, cpant \longrightarrow av', cpant' \quad s_t : av', cpant' \longrightarrow av, cpant}{\mathsf{while}\ b\ \mathsf{do}\ s_t : av, cpant \longrightarrow av', cpant'}\ \text{while}^{\text{ts}}_{\text{le}}$$

$$\frac{av, cpant \le av_0, cpant_0 \quad s : av_0, cpant_0 \longrightarrow av'_0, cpant'_0 \quad av'_0, cpant'_0 \le av', cpant'}{s : av, cpant \longrightarrow av', cpant'}\ \text{conseq}^{\text{ts}}_{\text{le}}$$

Figure 4.10: Type system for cpant analysis

pretype) only if it is already necessarily available there (thus in the intersection with the availability pretype). The rest of the rules mimic the rules of the available expressions and live variables analyses.

4.3.3 Type system for common subexpression elimination

The cpant expressions type system can now be used to perform common subexpression elimination. The rules of the optimization type system extend the cpant expressions type system; the rules are given in Figure 4.11.

Since additional variables need to be introduced into the program, we use an assignment $nv : \mathbf{AExp}^+ \to \mathbf{Var}_{\text{aux}}$ of a unique auxiliary program variable to every non-trivial arithmetical expression ($\mathbf{Var}_{\text{aux}}$ being an additional supply of program variables not available for normal programming). This is a way for two program points that will evaluate resp. reuse an expression value to agree on a variable that can safely (without the danger of a redefinition on the way) carry this value.

The main optimization is done in the rules for arithmetic expressions. The judgements for arithmetic expressions have the form $a : av, cpant \longrightarrow av', cpant' \hookrightarrow (nvd, a')$, where nvd is a sequence of assignments (auxiliary variable definitions emanating from a; the empty sequence is denoted by ϵ) and a' is an expression (the optimized version of a) (for boolean expressions, the judgements are similar). Note that optimizations need also be made "inside" expressions, since arithmetic subexpressions can be evaluated and later used within the same expression (for example, in the expression $x * y + x * y$, the subexpression $x * y$ does not have to be evaluated twice).

There are three rules for $+$: for the case where the compound expression is

$$\frac{}{n : av, cpant \longrightarrow av, cpant \hookrightarrow (\epsilon, n)}\ \text{num}_{\text{le}}^{\text{opt}} \qquad \frac{}{x : av, cpant \longrightarrow av, cpant \hookrightarrow (\epsilon, x)}\ \text{var}_{\text{le}}^{\text{opt}}$$

$$\frac{\begin{array}{c} a_0 + a_1 \in av \\ a_0 : av, cpant \longrightarrow av'', cpant'' \hookrightarrow (nvd_0, a_0') \quad a_1 : av'', cpant'' \longrightarrow av', cpant' \hookrightarrow (nvd_1, a_1') \end{array}}{a_0 + a_1 : av, cpant \cup \{a_0 + a_1\} \longrightarrow av' \cup \{a_0 + a_1\}, cpant' \hookrightarrow (nvd_0; nvd_1, nv(a_0 + a_1))}\ +1_{\text{le}}^{\text{opt}}$$

$$\frac{\begin{array}{c} a_0 + a_1 \in cpant' \qquad a_0 + a_1 \notin av \\ a_0 : av, cpant \longrightarrow av'', cpant'' \hookrightarrow (nvd_0, a_0') \quad a_1 : av'', cpant'' \longrightarrow av', cpant' \hookrightarrow (nvd_1, a_1') \end{array}}{\begin{array}{c} a_0 + a_1 : av, cpant \longrightarrow av' \cup \{a_0 + a_1\}, cpant' \\ \hookrightarrow (nvd_0; nvd_1; nv(a_0 + a_1) := a_0' + a_1', nv(a_0 + a_1)) \end{array}}\ +2_{\text{le}}^{\text{opt}}$$

$$\frac{\begin{array}{c} a_0 + a_1 \notin cpant' \qquad a_0 + a_1 \notin av \\ a_0 : av, cpant \longrightarrow av'', cpant'' \hookrightarrow (nvd_0, a_0') \quad a_1 : av'', cpant'' \longrightarrow av', cpant' \hookrightarrow (nvd_1, a_1') \end{array}}{a_0 + a_1 : av, cpant \longrightarrow av' \cup \{a_0 + a_1\}, cpant' \hookrightarrow (nvd_0; nvd_1, a_0' + a_1')}\ +3_{\text{le}}^{\text{opt}}$$

$$\frac{a_0 : av, cpant \longrightarrow av'', cpant'' \hookrightarrow (nvd_0, a_0') \quad a_1 : av'', cpant'' \longrightarrow av', cpant' \hookrightarrow (nvd_1, a_1')}{a_0 = a_1 : av, cpant \longrightarrow av', cpant' \hookrightarrow (nvd_0; nvd_1, a_0' = a_1')}\ =_{\text{le}}^{\text{opt}}$$

$$\frac{a : av, cpant \longrightarrow av', cpant' \hookrightarrow (nvd, a')}{x := a : av, cpant \longrightarrow av' \setminus mod(x), cpant' \hookrightarrow nvd; x := a'}\ :=_{\text{le}}^{\text{opt}}$$

$$\frac{}{\mathsf{skip} : av, cpant \longrightarrow av, cpant \hookrightarrow \mathsf{skip}}\ \text{skip}_{\text{le}}^{\text{opt}}$$

$$\frac{s_0 : av, cpant \longrightarrow av'', cpant'' \hookrightarrow s_0' \quad s_1 : av'', cpant'' \longrightarrow av', cpant' \hookrightarrow s_1'}{s_0; s_1 : av, cpant \longrightarrow av', cpant' \hookrightarrow s_0'; s_1'}\ \text{comp}_{\text{le}}^{\text{opt}}$$

$$\frac{\begin{array}{l} b : av, cpant \longrightarrow av'', cpant'' \hookrightarrow (nvd, b') \\ s_t : av'', cpant'' \longrightarrow av', cpant' \hookrightarrow s_t' \qquad s_f : av'', cpant'' \longrightarrow av', cpant' \hookrightarrow s_f' \end{array}}{\mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f : av, cpant \longrightarrow av', cpant' \hookrightarrow nvd; \mathsf{if}\ b'\ \mathsf{then}\ s_t'\ \mathsf{else}\ s_f'}\ \text{if}_{\text{le}}^{\text{opt}}$$

$$\frac{b : av, cpant \longrightarrow av', cpant' \hookrightarrow (nvd, b') \quad s_t : av', cpant' \longrightarrow av, cpant \hookrightarrow s_t'}{\mathsf{while}\ b\ \mathsf{do}\ s_t : av, cpant \longrightarrow av', cpant' \hookrightarrow nvd; \mathsf{while}\ b'\ \mathsf{do}\ (s_t'; nvd)}\ \text{while}_{\text{le}}^{\text{opt}}$$

$$\frac{av, cpant \leq av_0, cpant_0 \quad s : av_0, cpant_0 \longrightarrow av_0', cpant_0' \hookrightarrow s' \quad av_0', cpant_0' \leq av', cpant'}{s : av, cpant \longrightarrow av', cpant' \hookrightarrow s'}\ \text{conseq}_{\text{le}}^{\text{opt}}$$

Figure 4.11: Type system for common subexpression elimination

already available and can be replaced with the corresponding auxiliary variable (rule $+1^{\text{opt}}_{\text{le}}$), the case where the expression only becomes available, and is also cpant, so an auxiliary variable definition is introduced (rule $+2^{\text{opt}}_{\text{le}}$) and the case where an expression only becomes available, but is not cpant, so it is left as it is.

The judgements for statements are of the form $s : av, cpant \longrightarrow av', cpant' \hookrightarrow s'$, where s' is a statement (the optimized form of s). The rules for assignment, skip, composition and if-statements should be straightforward. The rule $\text{while}^{\text{opt}}_{\text{le}}$ for while-loops allows for reuse of expressions that are evaluated in the guard. Since a guard may be entered from two program points (the beginning of the loop and end of the loop body), the auxiliary variable definitions have to appear at both places.

The derivation in Figure 4.12 is an example of common subexpression elimination. At the beginning of the program, $p * q$ is available (this is just an assumption made). The expression $u * v$ becomes available after the first statement; the expression is used at three places. The information that it is used later on in the program reaches the first assignment (via the cpant expressions type). Therefore, the value of $u * v$ is recorded in the auxiliary variable. In the if-guard and then-branch, this expression is replaced with the new variable. In the else-branch, the expression $p * q$ is replaced with a variable holding the value of this expression.

Common subexpression elimination is sound in the sense that the optimized program and the original one simulate each other: at corresponding program points the states of the two programs agree on the normal program variables and, moreover, if some expression is in the cpant type, then its value in the state of the original program and the value of the corresponding auxiliary variable in the state of the optimized program coincide. Let $\sigma \sim_{cpant} \sigma'$ denote that two states σ (on the normal program variables $\mathbf{Var}$) and σ' (on the normal and auxiliary program variables $\mathbf{Var} + \mathbf{Var}_{\text{aux}}$) agree modulo $cpant \subseteq \mathbf{AExp}^+$ in the following sense: $\bigwedge_{x \in \mathbf{Var}} \sigma(x) = \sigma'(x) \wedge \bigwedge_{a \in cpant} [\![a]\!]\sigma = \sigma'(nv(a))$. The soundness theorem is:

Theorem 17 (Soundness of common subexpression elimination)
(o) If $av, cpant \leq av', cpant'$ *and* $\sigma \sim_{cpant} \sigma_*$*, then* $\sigma \sim_{cpant'} \sigma_*$*.*
(i) If $a : av, cpant \longrightarrow av', cpant' \hookrightarrow (nvd, a')$*,* $\sigma_* \succ\!nvd\!\rightarrow \sigma'_*$*, and* $\sigma \sim_{cpant} \sigma_*$*, then* $[\![a]\!]\sigma = [\![a']\!]\sigma'_*$ *and* $\sigma \sim_{cpant'} \sigma'_*$*.*
(ii) If $b : av, cpant \longrightarrow av', cpant' \hookrightarrow (nvd, b')$*,* $\sigma_* \succ\!nvd\!\rightarrow \sigma'_*$*, and* $\sigma \sim_{cpant} \sigma_*$*, then* $[\![b]\!]\sigma = [\![b']\!]\sigma'_*$ *and* $\sigma \sim_{cpant'} \sigma'_*$*.*
(iii) If $s : av, cpant \longrightarrow av', cpant' \hookrightarrow s'$ *and* $\sigma \sim_{cpant} \sigma_*$ *, then*
— $\sigma \succ\!s\!\rightarrow \sigma'$ *implies the existence of* σ'_* *such that* $\sigma' \sim_{cpant'} \sigma'_*$ *and* $\sigma_* \succ\!s'\!\rightarrow \sigma'_*$*,*
— $\sigma_* \succ\!s'\!\rightarrow \sigma'_*$ *implies the existence of* σ' *such that* $\sigma' \sim_{cpant'} \sigma'_*$ *and* $\sigma \succ\!s\!\rightarrow \sigma'$*.*

Proof. (o) As $av, cpant \leq av', cpant'$ implies $cpant \supseteq cpant'$, it is immediate that $\sigma \sim_{cpant} \sigma_*$ is a stronger statement than $\sigma \sim_{cpant'} \sigma_*$.

All of (i)-(iii) are proved by induction on the structure of the type derivation. We only prove the first half of (iii).

For (i), we use induction on the derivation of $a : av, cpant \longrightarrow av', cpant' \hookrightarrow (nvd, a')$. We assume that $\sigma_* \succ\!nvd\!\rightarrow \sigma'_*$ and $\sigma \sim_{cpant} \sigma_*$.

$$\dfrac{}{p*q : \{p*q, u*v\}, \{p*q\} \longrightarrow \{u*v, p*q\}, \emptyset \hookrightarrow (\epsilon, ptq)}$$

$$\dfrac{}{p*q+r : \{p*q, u*v\}, \{p*q\} \longrightarrow \{u*v, p*q, p*q+r\}, \emptyset \hookrightarrow (\epsilon, ptq+r)}$$

$$\dfrac{}{z := p*q+r : \{p*q, u*v\}, \{p*q\} \longrightarrow \{u*v, p*q, p*q+r\}, \emptyset \hookrightarrow z := ptq+r}$$

$$\dfrac{}{u*v : \{p*q, u*v\}, \{u*v\} \longrightarrow \{p*q, u*v\}, \emptyset \hookrightarrow (\epsilon, utv)}$$

$$\dfrac{}{p := u*v : \{p*q, u*v\}, \{u*v\} \longrightarrow \{u*v\}, \emptyset \hookrightarrow p := utv}$$

$$\dfrac{}{u*v : \{p*q, u*v\}, \{p*q, u*v\} \longrightarrow \{p*q, u*v\}, \{p*q, u*v\} \hookrightarrow (\epsilon, utv)}$$

$$\dfrac{}{u*v = c : \{p*q, u*v\}, \{p*q, u*v\} \longrightarrow \{p*q, u*v\}, \{p*q, u*v\} \hookrightarrow (\epsilon, utv = c)}$$

$$\dfrac{}{\begin{array}{r}\text{if } u*v = c \text{ then } p := u*v \text{ else } z := p*q+r : \{p*q, u*v\}, \{p*q, u*v\} \longrightarrow \{u*v\}, \emptyset \\ \hookrightarrow \text{if } utv = c \text{ then } p := utv \text{ else } z := ptq+r\end{array}}$$

$$\dfrac{}{z := 10 : \{p*q, u*v, u*v+z\}, \{p*q, u*v\} \longrightarrow \{p*q, u*v\}, \{p*q, u*v\} \hookrightarrow z := 10}$$

$$\dfrac{}{u*v : \{p*q\}, \{p*q\} \longrightarrow \{p*q, u*v\}, \{p*q, u*v\} \hookrightarrow (utv := u*v, utv)}$$

$$\dfrac{}{\begin{array}{r}u*v+z : \{p*q\}, \{p*q\} \longrightarrow \{p*q, u*v, u*v+z\}, \{p*q, u*v\} \\ \hookrightarrow (utv := u*v, utv+z)\end{array}}$$

$$\dfrac{}{\begin{array}{r}x := u*v+z : \{p*q\}, \{p*q\} \longrightarrow \{p*q, u*v, u*v+z\}, \{p*q, u*v\} \\ \hookrightarrow utv := u*v; x := utv+z\end{array}}$$

$$\dfrac{}{\begin{array}{r}x := u*v+z; z := 10 : \{p*q\}, \{p*q\} \longrightarrow \{p*q, u*v\}, \{p*q, u*v\} \\ \hookrightarrow utv := u*v; x := utv+z; z := 10\end{array}}$$

$$\dfrac{}{\begin{array}{r}x := u*v+z; z := 10; \text{if } u*v = c \text{ then } p := u*v \text{ else } z := p*q+r : \{p*q\}, \{p*q\} \longrightarrow \{u*v\}, \emptyset \\ \hookrightarrow utv := u*v; x := utv+z; z := 10; \text{if } utv = c \text{ then } p := utv \text{ else } z := ptq+r\end{array}}$$

Figure 4.12: An analysis and transformation of an example program

We keep in mind that nvd redefines no normal variables and no auxiliary variables for expressions in av and that all auxiliary variables of a' must be for expressions in av'.

We consider the following non-trivial cases.

- The type derivation is of the form

$$\dfrac{a_0 + a_1 \in av \quad \begin{array}{c}\vdots \\ a_0 : av, cpant \longrightarrow av'', cpant'' \\ \hookrightarrow (nvd_0, a_0')\end{array} \quad \begin{array}{c}\vdots \\ a_1 : av'', cpant'' \longrightarrow av', cpant' \\ \hookrightarrow (nvd_1, a_1')\end{array}}{a_0 + a_1 : av, cpant \cup \{a_0 + a_1\} \longrightarrow av' \cup \{a_0 + a_1\}, cpant' \hookrightarrow (nvd_0; nvd_1, nv(a_0 + a_1))}$$

  There must exist a state σ''_* such that $\sigma_* \succ nvd_0 \rightarrow \sigma''_*$ and $\sigma''_* \succ nvd_1 \rightarrow \sigma'_*$. The assumption $\sigma \sim_{cpant \cup \{a_0+a_1\}} \sigma_*$ implies $[\![a_0 + a_1]\!]\sigma = [\![nv(a_0 + a_1)]\!]\sigma_*$. As $a_0 + a_1 \in av$, we know that $nv(a_0 + a_1)$ is not modified by $nvd_0; nvd_1$, hence $[\![a_0 + a_1]\!]\sigma = [\![nv(a_0+a_1)]\!]\sigma_* = [\![nv(a_0+a_1)]\!]\sigma'_*$. The assumption $\sigma \sim_{cpant \cup \{a_0+a_1\}} \sigma_*$

also tell us that $\sigma \sim_{cpant} \sigma_*$, from where by the first induction hypothesis it follows that $\sigma \sim_{cpant''} \sigma''_*$ and further by the second induction hypothesis that $\sigma \sim_{cpant'} \sigma'_*$.

- The type derivation is of the form

$$\frac{\begin{array}{ccc} a_0 + a_1 \in cpant' & a_0 : av, cpant \longrightarrow av'', cpant'' & a_1 : av'', cpant'' \longrightarrow av', cpant' \\ a_0 + a_1 \notin av & \hookrightarrow (nvd_0, a'_0) & \hookrightarrow (nvd_1, a'_1) \end{array}}{a_0 + a_1 : av, cpant \longrightarrow av' \cup \{a_0 + a_1\}, cpant' \hookrightarrow (nvd_0; nvd_1; nv(a_0 + a_1) := a'_0 + a'_1, nv(a_0 + a_1))}$$

Again, we know that there must exist states σ''_* and σ'''_* such that $\sigma_* \succ nvd_0 \rightarrow \sigma''_*$, $\sigma''_* \succ nvd_1 \rightarrow \sigma'''_*$ and $\sigma'''_* \succ nv(a_0 + a_1) := a'_0 + a'_1 \rightarrow \sigma'_*$, so that $\sigma'_* = \sigma'''_*[nv(a_0 + a_1) \mapsto [\![a'_0 + a'_1]\!]\sigma'''_*]$. From $\sigma \sim_{cpant} \sigma_*$ by the two induction hypotheses it follows that $[\![a_0]\!]\sigma = [\![a'_0]\!]\sigma''_*$ and $\sigma \sim_{cpant''} \sigma''_*$ and further that $[\![a_1]\!]\sigma = [\![a'_1]\!]\sigma'''_*$ and $\sigma \sim_{cpant'} \sigma'''_*$. Since nvd_1 redefines no normal variables and no auxiliary variables for expressions in av'' and since all auxiliary variables of a'_0 are for expressions in av'', $[\![a'_0]\!]\sigma''_* = [\![a'_0]\!]\sigma'''_*$. Consequently, we see that $[\![a_0 + a_1]\!]\sigma = [\![a_0]\!]\sigma + [\![a_1]\!]\sigma = [\![a'_0]\!]\sigma'''_* + [\![a'_1]\!]\sigma'''_* = [\![a'_0 + a'_1]\!]\sigma'''_* = [\![nv(a_0 + a_1)]\!]\sigma'_*$. This in combination with $\sigma \sim_{cpant'} \sigma'''_*$ and the fact that the only change from σ'''_* to σ'_* concerns $nv(a_0 + a_1)$ also gives us that $\sigma \sim_{cpant'} \sigma'_*$.

- The type derivation is of the form

$$\frac{\begin{array}{ccc} a_0 + a_1 \notin cpant' & a_0 : av, cpant \longrightarrow av'', cpant'' & a_1 : av'', cpant'' \longrightarrow av', cpant' \\ a_0 + a_1 \notin av & \hookrightarrow (nvd_0, a'_0) & \hookrightarrow (nvd_1, a'_1) \end{array}}{a_0 + a_1 : av, cpant \longrightarrow av' \cup \{a_0 + a_1\}, cpant' \hookrightarrow (nvd_0; nvd_1, a'_0 + a'_1)}$$

We know that there must exist a state σ''_* such that $\sigma_* \succ nvd_0 \rightarrow \sigma''_*$ and $\sigma''_* \succ nvd_1 \rightarrow \sigma'_*$. From $\sigma \sim_{cpant} \sigma_*$ by the two induction hypotheses it follows that $[\![a_0]\!]\sigma = [\![a'_0]\!]\sigma''_*$ and $\sigma \sim_{cpant''} \sigma''_*$ and further that $[\![a_1]\!]\sigma = [\![a'_1]\!]\sigma'_*$ and $\sigma \sim_{cpant'} \sigma'_*$. Since nvd_1 redefines no normal variables and no auxiliary variables for expressions in av'' and since all auxiliary variables of a'_0 are for expressions in av'', $[\![a'_0]\!]\sigma''_* = [\![a'_0]\!]\sigma'_*$. Consequently, we see that $[\![a_0 + a_1]\!]\sigma = [\![a_0]\!]\sigma + [\![a_1]\!]\sigma = [\![a'_0]\!]\sigma'_* + [\![a'_1]\!]\sigma'_* = [\![a'_0 + a'_1]\!]\sigma'_*$.

The proof of (ii) is similar to that of (i). In particular, the case of $=^{\text{opt}}_{\text{le}}$ is analogous to the case of $+3^{\text{opt}}_{\text{le}}$.

(iii) We use induction on the structure of $s : av, cpant \longrightarrow av', cpant' \hookrightarrow s'$. We assume $\sigma \sim_{cpant} \sigma_*$ and $\sigma \succ s \rightarrow \sigma'$.

We consider the following cases:

- The type derivation is of the form

$$\frac{a : av, cpant \longrightarrow av', cpant' \hookrightarrow (nvd, a')}{x := a : av, cpant \longrightarrow av' \setminus mod(x), cpant' \hookrightarrow nvd; x := a'} :=^{\text{opt}}_{\text{le}}$$

The given semantic derivation must be of the form

$$\frac{}{\sigma \succ x := a \rightarrow \sigma[x \mapsto [\![a]\!]\sigma]}$$

so $\sigma' = \sigma[x \mapsto [\![a]\!]\sigma]$. Let σ''_* be the unique state such that $\sigma_* \succ nvd \rightarrow \sigma''_*$ and let $\sigma'_* = \sigma''_*[x \mapsto [\![a']\!]\sigma''_*]$, so we have a derivation of $\sigma''_* \succ x := a' \rightarrow \sigma'_*$. From the assumption $\sigma \sim_{cpant} \sigma_*$ by (i) we know that $[\![a]\!]\sigma = [\![a']\!]\sigma''_*$ and $\sigma \sim_{cpant'} \sigma''_*$. Consequently, $\sigma'(x) = [\![a]\!]\sigma = [\![a']\!]\sigma''_* = \sigma'_*(x)$. Further, since $cpant' \subseteq av' \setminus mod(x)$, no expression in $cpant'$ contains x. Accordingly, no expression in $cpant'$ can change its value during the assignment $x := a'$ taking from σ''_* to σ'_*. Therefore, from the knowledge that $\sigma \sim_{cpant'} \sigma''_*$ we may conclude that $\sigma' \sim_{cpant'} \sigma'_*$.

- The type derivation is of the form

$$\frac{\begin{array}{c} \vdots \\ b : av, cpant \longrightarrow av'', cpant'' \hookrightarrow (nvd, b') \end{array} \quad \begin{array}{c} \vdots \\ s_t : av'', cpant'' \longrightarrow av', cpant' \hookrightarrow s'_t \end{array} \quad \begin{array}{c} \vdots \\ s_f : av'', cpant'' \longrightarrow av', cpant' \hookrightarrow s'_f \end{array}}{\mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f : av, cpant \longrightarrow av', cpant' \hookrightarrow nvd; \mathsf{if}\ b'\ \mathsf{then}\ s'_t\ \mathsf{else}\ s'_f}\ \mathrm{if}^{\mathrm{opt}}_{\mathrm{le}}$$

We have that either $\sigma \models b$ or $\sigma \not\models b$. In the first case, the given semantic derivation is of the form

$$\frac{\sigma \models b \quad \begin{array}{c} \vdots \\ \sigma \succ s_t \rightarrow \sigma' \end{array}}{\sigma \succ \mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f \rightarrow \sigma'}$$

Let σ''_* be the unique state such that $\sigma_* \succ nvd \rightarrow \sigma''_*$. From $\sigma \sim_{cpant} \sigma_*$ by (ii) we learn that $[\![b]\!]\sigma = [\![b']\!]\sigma''_*$ and $\sigma \sim_{cpant''} \sigma''_*$. Thus we have the derivation

$$\frac{\sigma''_* \models b' \quad \begin{array}{c} \vdots\ IH \\ \sigma''_* \succ s'_t \rightarrow \sigma'_* \end{array}}{\sigma''_* \succ \mathsf{if}\ b'\ \mathsf{then}\ s'_t\ \mathsf{else}\ s'_f \rightarrow \sigma'_*}$$

By induction hypothesis we get that $\sigma' \sim_{cpant'} \sigma'_*$.

Similar reasoning holds for $\sigma \not\models b$.

- The type derivation is of the form

$$\frac{\begin{array}{c} \vdots \\ b : av, cpant \longrightarrow av', cpant' \hookrightarrow (nvd, b') \end{array} \quad \begin{array}{c} \vdots \\ s_t : av', cpant' \longrightarrow av, cpant \hookrightarrow s'_t \end{array}}{\mathsf{while}\ b\ \mathsf{do}\ s_t : av, cpant \longrightarrow av', cpant' \hookrightarrow nvd; \mathsf{while}\ b'\ \mathsf{do}\ (s'_t; nvd)}\ \mathrm{while}^{\mathrm{opt}}_{\mathrm{le}}$$

We also invoke structural induction on the given semantic derivation of $\sigma \succ \textsf{while}\ b\ \textsf{do}\ s_t \rightarrow \sigma'$. We have that either $\sigma \models b$ or $\sigma \not\models b$. In the first case, the given semantic derivation is of the form

$$\frac{\sigma \models b \quad \sigma \succ s_t \rightarrow \sigma'' \quad \sigma'' \succ \textsf{while}\ b\ \textsf{do}\ s_t \rightarrow \sigma'}{\sigma \succ \textsf{while}\ b\ \textsf{do}\ s_t \rightarrow \sigma'}$$

Let σ_*''' be the unique state such that $\sigma_* \succ nvd \rightarrow \sigma_*'''$. From $\sigma \sim_{cpant} \sigma_*$ by (ii) we infer that $[\![b]\!]\sigma = [\![b']\!]\sigma_*'''$ and $\sigma \sim_{cpant'} \sigma_*'''$. Thus we have the derivation

$$\frac{\sigma_*''' \models b' \quad \dfrac{\overset{\vdots\ \text{outer } IH}{\sigma_*''' \succ s_t' \rightarrow \sigma_*''} \quad \overset{\vdots\ \text{inner } IH}{\sigma_*'' \succ nvd \rightarrow \sigma_*''''}}{\sigma_*''' \succ s_t'; nvd \rightarrow \sigma_*''''} \quad \overset{\vdots\ \text{inner } IH}{\sigma_*'''' \succ \textsf{while}\ b'\ \textsf{do}\ (s_t'; nvd) \rightarrow \sigma_*'}}{\sigma_*''' \succ \textsf{while}\ b'\ \textsf{do}\ (s_t'; nvd) \rightarrow \sigma_*'}$$

Here the outer hypothesis applies thanks to $\sigma \sim_{cpant'} \sigma_*'''$ and ensures the existence of σ_*'' that also satisfies $\sigma'' \sim_{cpant} \sigma_*''$. Further, the inner hypothesis applies, taking care of the composition $nvd; \textsf{while}\ b'\ \textsf{do}\ (s_t'; nvd)$, and ensures the existence of σ_*' such that $\sigma' \sim_{cpant'} \sigma_*'$.

If $\sigma \not\models b$, then the semantic derivation must be

$$\frac{\sigma \not\models b}{\sigma \succ \textsf{while}\ b\ \textsf{do}\ s_t \rightarrow \sigma}$$

i.e., $\sigma' = \sigma$. Let state σ_*' be the unique state such that $\sigma_* \succ nvd \rightarrow \sigma_*'$. From $\sigma \sim_{cpant} \sigma_*$ by (ii) we know that $[\![b]\!]\sigma = [\![b']\!]\sigma_*'$ and $\sigma \sim_{cpant'} \sigma_*'$. Thus we have the derivation

$$\frac{\sigma_*' \not\models b'}{\sigma_*' \succ \textsf{while}\ b'\ \textsf{do}\ (s_t'; nvd) \rightarrow \sigma_*'}$$

and we also know that $\sigma \sim_{cpant'} \sigma_*'$.

- The type derivation is of the form

$$\frac{av, cpant \leq av_0, cpant_0 \quad \overset{\vdots}{s : av_0, cpant_0 \longrightarrow av_0', cpant_0' \hookrightarrow s'} \quad av_0', cpant_0' \leq av', cpant'}{s : av, cpant \longrightarrow av', cpant' \hookrightarrow s'}\ \text{conseq}_{\text{le}}^{\text{opt}}$$

We also have the given semantic judgement $\sigma \succ s \rightarrow \sigma'$. Since $cpant \supseteq cpant_0$, from $\sigma \sim_{cpant} \sigma_*$ we obtain that $\sigma \sim_{cpant_0} \sigma_*$. By the induction hypothesis, there must be a state σ_*' such that $\sigma_* \succ s' \rightarrow \sigma_*'$ and $\sigma' \sim_{cpant_0'} \sigma_*'$. Since $cpant_0' \supseteq cpant'$, we have that $\sigma' \sim_{cpant'} \sigma_*'$.

$\square$

To state the corresponding theorem about the Hoare logic, we define $P|_{cpant}$ to abbreviate $P \wedge \bigwedge_{a \in cpant} a = nv(a)$ and $P|^{cpant}$ to mean $\exists[v(a) \mid a \notin cpant](P[a/nv(a) \mid a \in cpant][v(a)/nv(a) \mid a \notin cpant])$.

The theorem is the following:

Theorem 18
(o) If $av, cpant \leq av', cpant'$, then $P|_{cpant} \models P|_{cpant'}$.
(i) If $a : av, cpant \longrightarrow av', cpant' \hookrightarrow (nvd, a')$, then
— $\{(P[a/w])|_{cpant}\}\, nvd\, \{(P|_{cpant'})[a'/w]\}$,
— $\{P\}\, nvd\, \{Q[a'/w]\}$ implies $P|^{cpant} \models (Q|^{cpant'})[a/w]$,
(ii) If $b : av, cpant \longrightarrow av', cpant' \hookrightarrow (nvd, b')$, then
— $\{(P[b/w])|_{cpant}\}\, nvd\, \{(P|_{cpant'})[b'/w]\}$; it follows that $\{P|_{cpant}\}\, nvd\, \{(b' \Rightarrow (b \wedge P)|_{cpant'}) \wedge (\neg b' \Rightarrow (\neg b \wedge P)|_{cpant'})\}$,
— $\{P\}\, nvd\, \{Q[b'/w]\}$ implies $P|^{cpant} \models (Q|^{cpant'})[b/w]$,
(iii) If $s : av, cpant \longrightarrow av', cpant' \hookrightarrow s'$, then
— $\{P\}\, s\, \{Q\}$ implies $\{P|_{cpant}\}\, s'\, \{Q|_{cpant'}\}$,
— $\{P\}\, s'\, \{Q\}$ implies $\{P|^{cpant}\}\, s\, \{Q|^{cpant'}\}$.

Proof. (o) Since $av, cpant \leq av', cpant'$ implies that $cpant \supseteq cpant'$, one gets $P|_{cpant} \models P|_{cpant'}$ by conjunction elimination.

Concerning (i)-(iii) we only prove the first halves. To save space, we do not show the entailment side conditions of the consequence rule.

(i) We use induction on the derivation of $a : av, cpant \longrightarrow av', cpant' \hookrightarrow (nvd, a')$. We restrict our attention to the following cases.

- The type derivation is of the form

$$\dfrac{a_0 + a_1 \in av \qquad \dfrac{\vdots}{a_0 : av, cpant \longrightarrow av'', cpant'' \hookrightarrow (nvd_0, a_0')} \qquad \dfrac{\vdots}{a_1 : av'', cpant'' \longrightarrow av', cpant' \hookrightarrow (nvd_1, a_1')}}{a_0 + a_1 : av, cpant \cup \{a_0 + a_1\} \longrightarrow av' \cup \{a_0 + a_1\}, cpant' \hookrightarrow (nvd_0; nvd_1, nv(a_0 + a_1))} +1_{\text{le}}^{\text{opt}}$$

We have the following derivation.

$$\dfrac{\dfrac{\dfrac{\vdots\ IH}{\{(P[nv(a_0 + a_1)/w])|_{cpant}\}\, nvd_0\, \{(P|[nv(a_0 + a_1)/w])_{cpant''})\}} \qquad \dfrac{\dfrac{\vdots\ IH}{\{(P[nv(a_0 + a_1)/w])|_{cpant''}\}\, nvd_1\, \{(P[nv(a_0 + a_1)/w])|_{cpant'}\}}}{}}{\{(P[nv(a_0 + a_1)/w])|_{cpant}\}\, nvd_0; nvd_1\, \{(P[nv(a_0 + a_1)/w])|_{cpant'}\}}}{\{(P[a_0 + a_1/w])|_{cpant \cup \{a_0 + a_1\}}\}\, nvd_0; nvd_1\, \{(P|_{cpant'})[nv(a_0 + a_1)/w]\}}$$

The entailment $(P[a_0 + a_1/w])|_{cpant \cup \{a_0 + a_1\}} \models (P[nv(a_0 + a_1)/w])|_{cpant}$ holds as $(P[a_0 + a_1/w])|_{cpant \cup \{a_0 + a_1\}} \models (P[a_0 + a_1/w])|_{cpant} \wedge nv(a_0 + a_1) = a_0 + a_1 \models (P[nv(a_0 + a_1)/w])|_{cpant}$ by substitution of equals for equals.

- The type derivation is of the form

$$\frac{\begin{array}{c} a_0+a_1\in cpant' \\ a_0+a_1\notin av \end{array} \quad \begin{array}{c}\vdots\\ a_0 : av, cpant \longrightarrow av'', cpant'' \\ \hookrightarrow (nvd_0, a_0') \end{array} \quad \begin{array}{c}\vdots\\ a_1 : av'', cpant'' \longrightarrow av', cpant' \\ \hookrightarrow (nvd_1, a_1')\end{array}}{a_0+a_1 : av, cpant \longrightarrow av'\cup\{a_0+a_1\}, cpant' \hookrightarrow (nvd_0; nvd_1; nv(a_0+a_1) := a_0'+a_1', nv(a_0+a_1))}$$

We have the following derivation.

$$\frac{\dfrac{}{\{(P[nv(a_0+a_1)/w]|_{cpant'})[a_0'+a_1'/nv(a_0+a_1)]\}\, nv(a_0+a_1) := a_0'+a_1' \,\{(P[nv(a_0+a_1)/w])|_{cpant'}\}}}{\{(P[a_0'+a_1'/w])|_{cpant'} \wedge a_0+a_1 = a_0'+a_1'\}\, nv(a_0+a_1) := a_0'+a_1' \,\{(P|_{cpant'})[nv(a_0+a_1)/w]\}}$$

$$\frac{\begin{array}{c}\vdots\ IH\\ \{((P[a_0'+w_1/w]\wedge a_0=a_0'\wedge a_1=w_1)[a_1/w_1])|_{cpant''}\}\\ nvd_1\\ \{((P[a_0'+w_1/w]\wedge a_0=a_0'\wedge a_1=w_1)|_{cpant'})[a_1'/w_1]\}\end{array}}{\{(P[a_0'+a_1/w]\wedge a_0=a_0')|_{cpant''}\}\, nvd_1 \,\{(P[a_0'+a_1'/w]\wedge a_0=a_0'\wedge a_1=a_1')|_{cpant'}\}}$$

$$\frac{\dfrac{\dfrac{\begin{array}{c}\vdots\ IH\\ \{((P[w_0+a_1/w]\wedge a_0=w_0)[a_0/w_0])|_{cpant}\}\\ nvd_0\\ \{((P[w_0+a_1/w]\wedge a_0=w_0)|_{cpant''})[a_0'/w_0]\}\end{array}}{\{(P[a_0+a_1/w])|_{cpant}\}\, nvd_0 \,\{(P[a_0'+a_1/w]\wedge a_0=a_0')|_{cpant''}\}}}{\{(P[a_0+a_1/w])|_{cpant}\}\, nvd_0; nvd_1 \,\{(P[a_0'+a_1'/w])|_{cpant'}\wedge a_0+a_1=a_0'+a_1'\}}}{\{(P[a_0+a_1/w])|_{cpant}\}\, nvd_0; nvd_1; nv(a_0+a_1) := a_0'+a_1' \,\{(P|_{cpant'})[nv(a_0+a_1)/w]\}}$$

In this derivation, w_0, w_1 are picked fresh. The entailment $(P[a_0'+a_1'/w])|_{cpant'} \wedge a_0+a_1 = a_0'+a_1' \models ((P[nv(a_0+a_1)/w])|_{cpant'})[a_0'+a_1'/nv(a_0+a_1)]$ holds because $(P[nv(a_0+a_1)/w])|_{cpant'} \Leftrightarrow (P[nv(a_0+a_1)/w])|_{cpant'\setminus\{a_0+a_1\}} \wedge a_0+a_1 = nv(a_0+a_1)$, so $(P[nv(a_0+a_1)/w]|_{cpant'})[a_0'+a_1'/nv(a_0+a_1)] \Leftrightarrow (P[a_0'+a_1'/w])|_{cpant'\setminus\{a_0+a_1\}} \wedge a_0+a_1 = a_0'+a_1'$.

- The type derivation is of the form

$$\frac{\begin{array}{c} a_0+a_1\notin cpant' \\ a_0+a_1\notin av \end{array} \quad \begin{array}{c}\vdots\\ a_0 : av, cpant \longrightarrow av'', cpant'' \\ \hookrightarrow (nvd_0, a_0') \end{array} \quad \begin{array}{c}\vdots\\ a_1 : av'', cpant'' \longrightarrow av', cpant' \\ \hookrightarrow (nvd_1, a_1')\end{array}}{a_0+a_1 : av, cpant \longrightarrow av'\cup\{a_0+a_1\}, cpant' \hookrightarrow (nvd_0; nvd_1; a_0'+a_1')}\ +3^{\text{opt}}_{\text{le}}$$

We have this Hoare derivation:

$$\frac{\begin{array}{c}\vdots\ IH\\ \{((P[a_0'+w_1/w])[a_1/w_1])|_{cpant''}\}\, nvd_1 \,\{((P[a_0'+w_1/w])|_{cpant'})[a_1'/w_1]\}\end{array}}{\{(P[a_0'+a_1/w])|_{cpant''}\}\, nvd_1 \,\{(P[a_0'+a_1'/w])|_{cpant'}\}}$$

$$\frac{\dfrac{\begin{array}{c}\vdots\ IH\\ \{((P[w_0+a_1/w])[a_0/w_0])|_{cpant}\}\, nvd_0 \,\{((P[w_0+a_1/w])|_{cpant''})[a_0'/w_0]\}\end{array}}{\{(P[a_0+a_1/w])|_{cpant}\}\, nvd_0 \,\{(P[a_0'+a_1/w])|_{cpant''}\}}}{\{(P[a_0+a_1/w])|_{cpant}\}\, nvd_0; nvd_1 \,\{(P|_{cpant'})[a_0'+a_1'/w]\}}$$

The entailments are justified as in the previous case.

The proof of (ii) is similar to that of (i). In particular, the case of $=_{\mathrm{le}}^{\mathrm{opt}}$ is analogous to the case of $+3_{\mathrm{le}}^{\mathrm{opt}}$. Once the main statement has been established, the consequence $\{P|_{cpant}\}\, nvd\, \{(b' \Rightarrow (b \wedge P)|_{cpant'}) \wedge (\neg b' \Rightarrow (\neg b \wedge P)|_{cpant'})\}$ follows from the instance

$$\begin{array}{c}\{(((w \Rightarrow (b \wedge P)) \wedge (\neg w \Rightarrow (\neg b \wedge P)))[b/w])|_{cpant}\} \\ nvd \\ \{(((w \Rightarrow (b \wedge P)) \wedge (\neg w \Rightarrow (\neg b \wedge P)))|_{cpant'})[b'/w]\}\end{array}$$

by the entailments $P|_{cpant} \models ((b \Rightarrow (b \wedge P)) \wedge (\neg b \Rightarrow (\neg b \wedge P))|_{cpant} \equiv (((w \Rightarrow (b \wedge P)) \wedge (\neg w \Rightarrow (\neg b \wedge P)))[b/w])|_{cpant}$ and $(((w \Rightarrow (b \wedge P)) \wedge (\neg w \Rightarrow (\neg b \wedge P)))|_{cpant'})[b'/w] \models ((w \Rightarrow (b \wedge P)|_{cpant'}) \wedge (\neg w \Rightarrow (\neg b \wedge P)|_{cpant'}))[b/w] \equiv (b' \Rightarrow (b \wedge P)|_{cpant'}) \wedge (\neg b' \Rightarrow (\neg b \wedge P)|_{cpant'})$.

(iii) We use induction on the derivation of $s : av, cpant \longrightarrow av', cpant' \hookrightarrow s'$. We consider the following cases.

- The type derivation is

$$\frac{a : av, cpant \longrightarrow av', cpant' \hookrightarrow (nvd; a')}{x := a : av, cpant \longrightarrow av' \setminus mod(x), cpant' \hookrightarrow nvd; x := a'}\ :=_{\mathrm{le}}^{\mathrm{opt}}$$

  The given Hoare derivation must be of the form

$$\frac{}{\{P[a/x]\}\, x := a\, \{P\}}$$

  This translates into the following Hoare derivation

$$\dfrac{\dfrac{\begin{array}{c}\vdots\ \text{(ii)} \\ \{(P[w/x][a/w])|_{cpant}\}\, nvd\, \{(P[w/x])|_{cpant'}[a'/w]\}\end{array}}{\{(P[a/x])|_{cpant}\}\, nvd\, \{P|_{cpant'}[w/x][a'/w]\}} \quad \dfrac{\dfrac{}{\{P|_{cpant'}[a'/x]\}\, x := a'\, \{P|_{cpant'}\}}}{\{P|_{cpant'}[w/x][a'/w]\}\, x := a'\, \{P|_{cpant'}\}}}{\{P[a/x]|_{cpant}\}\, nvd; x := a'\, \{P|_{cpant'}\}}$$

  We have $(P[w/x])|_{cpant'}[a'/w] \models P|_{cpant'}[w/x][a'/w]$, since $cpant' \subseteq av' \setminus mod(x)$, so there are no expressions with x as a free variable in $cpant'$.

- The type derivation is of the form

$$\frac{\begin{array}{c}\vdots \\ b : av, cpant \longrightarrow av'', cpant'' \\ \hookrightarrow (nvd, b')\end{array} \quad \begin{array}{c}\vdots \\ s_t : av'', cpant'' \longrightarrow av', cpant' \\ \hookrightarrow s'_t\end{array} \quad \begin{array}{c}\vdots \\ s_f : av'', cpant'' \longrightarrow av', cpant' \\ \hookrightarrow s'_f\end{array}}{\mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f : av, cpant \longrightarrow av', cpant' \hookrightarrow nvd; \mathsf{if}\ b'\ \mathsf{then}\ s'_t\ \mathsf{else}\ s'_f}\ \mathrm{if}_{\mathrm{le}}^{\mathrm{opt}}$$

  and the given Hoare derivation is of the form

$$\frac{\begin{array}{c}\vdots \\ \{b \wedge P\}\, s_t\, \{Q\}\end{array} \quad \begin{array}{c}\vdots \\ \{\neg b \wedge P\}\, s_f\, \{Q\}\end{array}}{\{P\}\, \mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f\, \{Q\}}\ .$$

Let $P' =_{\mathrm{df}} b' \Rightarrow (b \wedge P)|_{cpant''} \wedge \neg b' \Rightarrow (\neg b \wedge P)|_{cpant''}$. We can make the following derivation:

$$\dfrac{\dfrac{\vdots\ (ii)}{\{P|_{cpant}\}\, nvd\, \{P'\}} \quad \dfrac{\dfrac{\dfrac{\vdots\ IH}{\{(b \wedge P)|_{cpant''}\}\, s_t\, \{Q|_{cpant'}\}}}{\{b' \wedge P'\}\, s_t\, \{Q|_{cpant'}\}} \quad \dfrac{\dfrac{\vdots\ IH}{\{(\neg b \wedge P)|_{cpant''}\}\, s_f\, \{Q|_{cpant'}\}}}{\{\neg b' \wedge P'\}\, s_f\, \{Q|_{cpant'}\}}}{\{P'\}\, \mathsf{if}\ b'\ \mathsf{then}\ s'_t\ \mathsf{else}\ s'_f\, \{Q|_{cpant'}\}}}{\{P|_{cpant}\}\, nvd; \mathsf{if}\ b'\ \mathsf{then}\ s'_t\ \mathsf{else}\ s'_f\, \{Q|_{cpant'}\}}\ .$$

- The type derivation is of the form

$$\dfrac{\dfrac{\vdots}{b : av, cpant \longrightarrow av', cpant' \hookrightarrow (nvd; b')} \quad \dfrac{\vdots}{s_t : av', cpant' \longrightarrow av, cpant \hookrightarrow s'_t}}{\mathsf{while}\ b\ \mathsf{do}\ s_t : av, cpant \longrightarrow av', cpant' \hookrightarrow nvd; \mathsf{while}\ b'\ \mathsf{do}\ (s'_t; nvd)}\ \mathrm{while}^{\mathrm{opt}}_{\mathrm{le}}$$

  The given Hoare derivation must be of the form

$$\dfrac{\dfrac{\vdots}{\{b \wedge P\}\, s_t\, \{P\}}}{\{P\}\, \mathsf{while}\ b\ \mathsf{do}\ s_t\, \{\neg b \wedge P\}}$$

  Let $P' =_{\mathrm{df}} b' \Rightarrow (b \wedge P)|_{cpant'} \wedge \neg b' \Rightarrow (\neg b \wedge P)|_{cpant'}$. The transformed Hoare derivation is

$$\dfrac{\dfrac{\vdots\ (ii)}{\{P|_{cpant}\}\, nvd\, \{P'\}} \quad \dfrac{\dfrac{\dfrac{\dfrac{\vdots\ IH}{\{(b \wedge P)|_{cpant'}\}\, s'_t\, \{P|_{cpant}\}}}{\{b' \wedge P'\}\, s'_t\, \{P|_{cpant}\}} \quad \dfrac{\vdots\ (ii)}{\{P|_{cpant}\}\, nvd\, \{P'\}}}{\{b' \wedge P'\}\, s'_t; nvd\, \{P'\}}}{\{P'\}\, \mathsf{while}\ b'\ \mathsf{do}\ (s'_t; nvd)\, \{\neg b' \wedge P'\}}}{\{P|_{cpant}\}\, nvd; \mathsf{while}\ b'\ \mathsf{do}\ (s'_t; nvd)\, \{(\neg b \wedge P)|_{cpant'}\}}$$

□

A Hoare proof for the program analyzed in Figure 4.12 is given in Figure 4.13. A Hoare proof for the optimized program is given in Figure 4.14. As can be seen from the example, the precondition of the program needs to be strengthened according to the given pretype. Similarly, assertions for the intermediate program points are also strengthened, according to the types.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\{\top\}\ x := u*v+z\ \{\top\}} \quad \dfrac{}{\{\top\}\, z := 10\, \{\top\}}}{\{\top\}\ x := u*v+z; z := 10\ \{\top\}} \qquad \dfrac{\dfrac{\dfrac{}{\{u*v = u*v\}\, p := u*v\, \{p = u*v\}}}{\{u*v = c \wedge \top\}\, p := u*v\, \{p = u*v \vee z = p*q+r\}} \quad \dfrac{\dfrac{}{\{p*q+r = p*q+r\}\, z := p*q+r\, \{z = p*q+r\}}}{\{u*v \neq c \wedge \top\}\, z := p*q+r\, \{p = u*v \vee z = p*q+r\}}}{\{\top\}\ \text{if } u*v = c \text{ then } p := u*v \text{ else } z := p*q+r\ \{p = u*v \vee z = p*q+r\}}}{\{\top\}\ \begin{array}{l} x := u*v+z; z := 10; \\ \text{if } u*v = c \text{ then } p := u*v \text{ else } z := p*q+r \end{array}\ \{p = u*v \vee z = p*q+r\}}$$

Figure 4.13: A proof of the example program

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\{ptq = p*q\}\, utv := u*v\, \left\{\begin{array}{l} utv = u*v \wedge \\ ptq = p*q \end{array}\right\}} \quad \dfrac{}{\left\{\begin{array}{l} utv = u*v \wedge \\ ptq = p*q \end{array}\right\} x := utv + z \left\{\begin{array}{l} utv = u*v \wedge \\ ptq = p*q \end{array}\right\}}}{\{ptq = p*q\}\ utv := u*v; x := utv + z\ \left\{\begin{array}{l} utv = u*v \wedge \\ ptq = p*q \end{array}\right\}} \quad \dfrac{}{\left\{\begin{array}{l} utv = u*v \wedge \\ ptq = p*q \end{array}\right\} z := 10 \left\{\begin{array}{l} utv = u*v \wedge \\ ptq = p*q \end{array}\right\}}}{\{ptq = p*q\}\ utv := u*v; x := utv + z; z := 10\ \left\{\begin{array}{l} utv = u*v \wedge \\ ptq = p*q \end{array}\right\}} \quad \dfrac{\dfrac{\dfrac{\dfrac{}{\{utv = u*v\}\, p := utv\, \{p = u*v\}}}{\{u*v = u*v \wedge utv = u*v\}\, p := utv\, \{p = u*v\}}}{\{u*v = c \wedge utv = u*v\}\, p := utv\, \{p = u*v \vee z = p*q+r\}} \quad \dfrac{\dfrac{\dfrac{}{\{ptq + r = p*q+r\}\, z := ptq + r\, \{z = p*q+r\}}}{\{p*q+r = p*q+r \wedge ptq = p*q\}\, z := ptq + r\, \{z = p*q+r\}}}{\{u*v \neq c \wedge ptq = p*q\}\, z := ptq + r\, \{p = u*v \vee z = p*q+r\}}}{\{utv = u*v \wedge ptq = p*q\}\ \text{if } utv = c \text{ then } p := utv \text{ else } z := ptq + r\ \{p = u*v \vee z = p*q+r\}}}{\{ptq = p*q\}\ \begin{array}{l} utv := u*v; x := utv + z; z := 10; \\ \text{if } utv = c \text{ then } p := utv \text{ else } z := ptq + r \end{array}\ \{p = u*v \vee z = p*q+r\}}$$

Figure 4.14: Transformed proof

4.4 Partial redundancy elimination

In this section we look at a particularly complex optimization called partial redundancy elimination (PRE). PRE is a widely used compiler optimization that eliminates computations that are redundant on some but not necessarily all computation paths of a program. This optimization is notoriously tricky and has been extensively studied since it was invented by Morel and Renvoise [44]. There is no single canonical algorithm for performing the optimization. Instead, there is a plethora of subtly different ones. The clearest formulations by Paleri et al. [50] and Xue and Knoop [69] are based on four unidirectional dataflow analyses.

As a case study, the optimization is interesting in several aspects. As already said it is a highly nontrivial optimization. It is also interesting in the sense that it modifies program structure by inserting new nodes into the edges of the control flow graph. This makes automatic proof transformation potentially more difficult.

The language used While using the control flow graph based representations of program is common in the literature dealing with PRE, we will work directly with While programs like we did for dead code elimination and common subexpression elimination. However, to simplify the presentation we allow expressions to contain at most one operator. This is to mimic the standard algorithms for partial redundancy elimination, which do no consider deep expressions. As was shown in previous sections, this is an inessential restriction: with the help of just a little more infrastructure we could also instead treat optimizations handling deep expressions directly.

The grammar for arithmetic and boolean expressions is thus the following ($l \in$ **Lit** are the literals):

$$\begin{array}{rcl} l & ::= & x \mid n \\ a & ::= & l \mid l_0 + l_1 \mid l_0 * l_1 \mid \ldots \\ b & ::= & l_0 = l_1 \mid l_0 \leq l_1 \mid \ldots \end{array}$$

We write $\mathbf{AExp}^+$ for the set $\mathbf{AExp} \setminus \mathbf{Lit}$ of nontrivial arithmetic expressions. The rest of the language remains unchanged.

4.4.1 Simple PRE

PRE is an optimization to avoid computations of expressions that are redundant on some but not necessarily all computation paths of the program. Elimination of these computations happens at the expense of precomputing expressions and using auxiliary variables to remember their values. In some sense PRE is similar to CSE. The main difference between the two lies in the fact that while CSE only saves computed values at their evaluation places as needed, PRE tries to actively

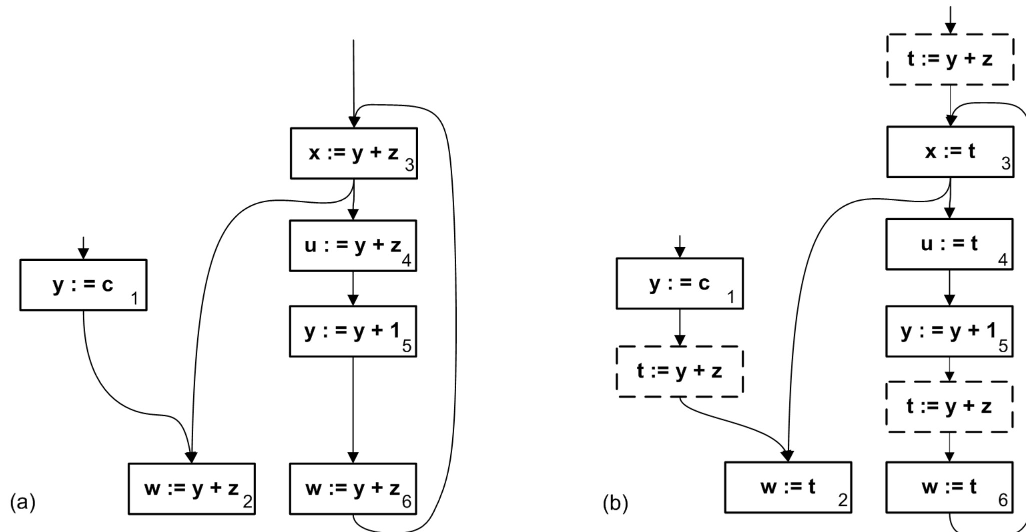


Figure 4.15: Example application of PRE

place new computations into the control flow graph to reduce the total number of evaluations needed by the program. As such, PRE subsumes CSE.

An example application of partial redundancy elimination is shown in Figure 4.15 where the graph in Figure 4.15(a) represents the original program and the graph in Figure 4.15(b) represents the program after partial redundancy elimination optimization. The computations of $y + z$ in nodes 2 and 3 are partially redundant in the original program and can be eliminated, and likewise the fully redundant computation in node 4. The result of the computation of $y + z$ in node 6 can be saved into an auxiliary variable t (thus a new node is added in front of node 6). Additional computations of $y + z$ are inserted into the edge leading from node 1 to node 2 and the edge entering node 3. This is the optimal arrangement of computations, since there are two fewer evaluations of $y + z$ in the loop and one fewer on the path leading from node 3 to 2. Furthermore, the number of evaluations of the expression has not increased on any path through the program.

In this section we scrutinize a simplified version of PRE that is more conservative than full PRE. Although powerful already, it does not eliminate all partial redundancies, but is more easily presentable, relying on two dataflow analyses. The example program in Figure 4.15(a) can be fully optimized by simple PRE. Its deficiencies will be discussed in Section 4.4.2, where the description of full PRE is given.

The two dataflow analyses are a backward anticipability analysis and a forward nonstandard, *conditional* partial availability analysis that uses the results of the anticipability analysis. The anticipability analysis computes for each program point which nontrivial arithmetic expressions will be evaluated on all outgoing paths before

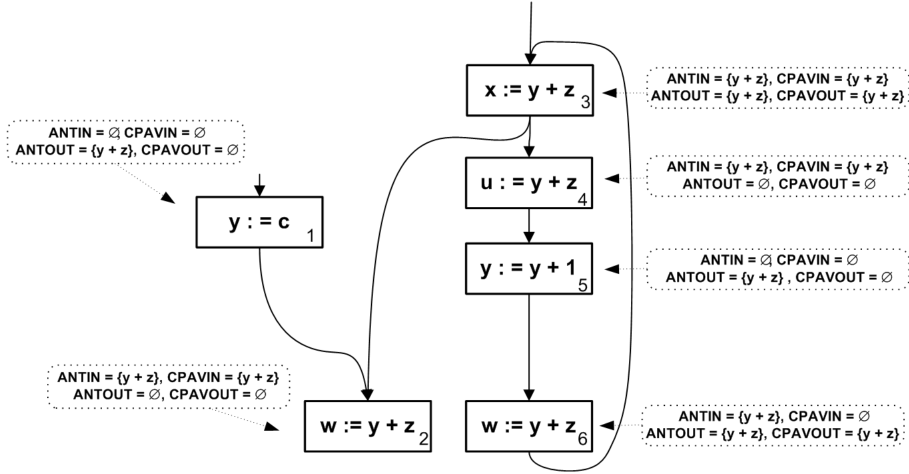


Figure 4.16: Property annotations on example program

any of their operands are modified. The partial availability analysis computes which expressions have already been evaluated and later not modified on some paths to a program point where that expression is anticipable. (As such, it is symmetric to common subexpression elimination.)

There are two possible optimizations for assignments. If we know that an expression is anticipable after an assignment, it means that the expression will definitely be evaluated later on in the program, so an auxiliary variable can be introduced, to carry the result of the evaluation. If we know that an expression is conditionally partially available before the assignment, we can assume it has already been computed and replace the expression with the auxiliary variable holding its value. If neither case holds, we leave the assignment unchanged. To make a conditionally partially available expression fully available in the optimized program, we must perform code motion, i.e., insert evaluations of expressions into edges where they are not partially available at the beginning-point but are partially available at the end-point.

The standard-style description of the algorithm relies on the properties $ANTIN$, $ANTOUT$, $CPAVIN$, $CPAVOUT$, MOD, $EVAL$. The global properties $ANTIN_i$ ($ANTOUT_i$) denote anticipability of nontrivial arithmetic expressions at the entry (exit) of node i. Similarly $CPAVIN_i$ and $CPAVOUT_i$ denote conditional partial availability. The local property MOD_i denotes the set of expressions whose value might be modified at node i whereas $EVAL_i$ denotes the set of nontrivial expressions which are evaluated at node i. The standard inequations for the analyses for CFGs

are given below (s and f correspond to the start and finish nodes of the whole CFG).

$$
\begin{array}{rcl}
ANTOUT_i & \subseteq & \begin{cases} \emptyset & \text{if } i = f \\ \bigcap_{j \in succ(i)} ANTIN_j & \text{otherwise} \end{cases} \\
ANTIN_i & \subseteq & (ANTOUT_i \setminus MOD_i) \cup EVAL_i \\
CPAVIN_i & \supseteq & \begin{cases} \emptyset & \text{if } i = s \\ \bigcup_{j \in pred(i)} CPAVOUT_i & \text{otherwise} \end{cases} \\
CPAVOUT_i & \supseteq & ((CPAVIN_i \cup EVAL_i) \setminus MOD_i) \\
 & \cap & ANTOUT_i \\
CPAVIN_i & \subseteq & ANTIN_i \\
CPAVOUT_i & \subseteq & ANTOUT_i
\end{array}
$$

We use inequalities here instead of equalities to be more in line with our type system, since the type system will accept all valid analysis results, not only the strongest one. Also note that the last inequalities do not correspond to transfer functions. Instead they state a domain restriction on conditional partial availability sets.

Annotations corresponding to these inequalities are given in Figure 4.16, leading to the optimization in Figure 4.15. For example, the expression evaluations in nodes 2, 3 and 4 can be replaced, since the expressions are partially available at the entry of the node (*CPAVIN*) and will be made fully available by the optimizations. At node 6, the expression is anticipable at the exit, and not available at the entry, thus this is the place where the result should be precomputed and saved in a temporary. For this, a new node to save the result of the computation is inserted before node 6. Edge splitting happens between nodes 1 and 2, and at the entry edge into node 3. At the exit of node 1, the expression is not partially available, but is at the entry to node 2. Thus the edge needs to be split, and a new node computing the expression added, thus making it *fully* available. The same happens to the entry edge of node 3.

Type system for simple PRE

We now present the two analyses as type systems. Types and subtyping for anticipability are sets of nontrivial arithmetic expressions and set inclusion, i.e, $(\mathcal{P}(\mathbf{AExp}^+), \subseteq)$. A program point has type $ant \in \mathcal{P}(\mathbf{AExp}^+)$ if all the expressions in ant are anticipable, i.e., on all paths from that program point, there will be a use of the expression before any of its operands are modified. Subtyping is set inclusion, i.e., $ant \leq ant'$ iff $ant \subseteq ant'$. Typing judgements $s : ant \longrightarrow ant'$ associate a statement with a pre- and posttype pair, stating that, if at the end of a statement s the expressions in ant' are anticipable, then at the beginning the expressions in ant must be anticipable. The typing rules are given in Figure 4.17. We use $eval(a)$ to denote the set $\{a\}$, if a is a nontrivial expression, and $\emptyset$ otherwise, and $mod(x)$ to denote the set of nontrivial expressions containing x, i.e., $mod(x) =_{\mathrm{df}} \{a \mid x \in FV(a)\}$. The assignment rule states that the assignment to

x kills all expressions containing x and at the same time the expression assigned becomes anticipable, if nontrivial. To type an if-statement the pre- and posttypes for both branches have to match. For a while loop, some type must be invariant for the loop body.

The anticipability analysis gives us the information about where it is definitely profitable to precompute expressions. Intuitively, they should be precomputed where they first become available and are anticipable, and reused where they are already available. The second analysis, the conditional partial availability analysis, propagates this information forward in the control flow graph. As it depends on the anticipability analysis, we need to combine the two in the type system. For the combined type system, a type is a pair $(ant, cpav) \in \mathcal{P}(\mathbf{AExp}^+) \times \mathcal{P}(\mathbf{AExp}^+)$ satisfying the constraint $cpav \subseteq ant$, where ant is an anticipability type and $cpav$ is a conditional partial availability type. Subtyping $\leq$ is pointwise set inclusion, i.e. $(ant, cpav) \leq (ant', cpav')$ iff $ant \subseteq ant'$ and $cpav \subseteq cpav'$. Typing judgements take the form $s : ant, cpav \longrightarrow ant', cpav'$. The intended meaning of the added conditional partial availability component here is that, if the expressions in $cpav$ are conditionally partially available before running s, then expressions in $cpav'$ may be conditionally partially available after running the program.

The typing rules of the combined type system are given in Figure 4.18. The rules for assignment now have the conditional partial availability component. An expression is conditionally partially available in the posttype of an assignment if it is so already in the pretype or is evaluated by the assignment and the assignment does not modify any of its operands. Additionally, the expression is only declared conditonally partially available if it is actually anticipable (worth to be precomputed), thus the intersection with the anticipability type.

The optimization component of the type system is shown in Figure 4.19. Definitions of auxiliary variables can be introduced in two places, before assignments (if the necessary conditions are met) and at subsumptions. An already computed value is used if an expression is conditionally partially available (rule $:=_{3\mathrm{pre}}$). If it is not, but is anticipable (will definitely be used), and the assignment does not change the value of the expression, then the result of evaluating it is recorded in the auxiliary variable for that expression (rule $:=_{2\mathrm{pre}}$). The auxiliary function nv delivers a unique new auxiliary variable for every nontrivial arithmetic expression, just like it did for CSE. Edge splitting is performed by the subsumption rule, which introduces auxiliary variable definitions when there is shrinking or growing of types (this typically happens at the beginning of loops and at the end of conditional branches and loop bodies).

Semantic soundness and improvement

Again, we take soundness to mean that an original program and its optimized version simulate each other up to a similarity relation $\sim$ on states, indexed by conditional partial availability types of program points (remember that for CSE, it was conditional partial anticipability types).

$$\frac{}{x := a : ant' \setminus mod(x) \cup eval(a) \longrightarrow ant'} \qquad \frac{}{\mathsf{skip} : ant \longrightarrow ant}$$

$$\frac{s_0 : ant \longrightarrow ant'' \quad s_1 : ant'' \longrightarrow ant'}{s_0; s_1 : ant \longrightarrow ant'} \qquad \frac{s_t : ant \longrightarrow ant' \quad s_f : ant \longrightarrow ant'}{\mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f : ant \longrightarrow ant'}$$

$$\frac{s_t : ant \longrightarrow ant}{\mathsf{while}\ b\ \mathsf{do}\ s_t : ant \longrightarrow ant} \qquad \frac{ant \leq ant_0 \quad s : ant_0 \longrightarrow ant'_0 \quad ant'_0 \leq ant'}{s : ant \longrightarrow ant'}$$

Figure 4.17: Type system for anticipability

$$\frac{}{x := a : ant' \setminus mod(x) \cup eval(a), cpav \longrightarrow ant', (cpav \cup eval(a) \setminus mod(x)) \cap ant'}$$

$$\frac{}{\mathsf{skip} : ant, cpav \longrightarrow ant, cpav} \qquad \frac{s_0 : ant, cpav \longrightarrow ant'', cpav'' \quad s_1 : ant'', cpav'' \longrightarrow ant', cpav'}{s_0; s_1 : ant, cpav \longrightarrow ant', cpav'}$$

$$\frac{s_t : ant, cpav \longrightarrow ant', cpav' \quad s_f : ant, cpav \longrightarrow ant', cpav'}{\mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f : ant, cpav \longrightarrow ant', cpav'} \qquad \frac{s_t : ant, cpav \longrightarrow ant, cpav}{\mathsf{while}\ b\ \mathsf{do}\ s_t : ant, cpav \longrightarrow ant, cpav}$$

$$\frac{ant, cpav \leq ant_0, cpav_0 \quad s : ant_0, cpav_0 \longrightarrow ant'_0, cpav'_0 \quad ant'_0, cpav'_0 \leq ant', cpav'}{s : ant, cpav \longrightarrow ant', cpav'}$$

Figure 4.18: Type system for the underlying analyses of simple PRE

Let $\sigma \sim_{cpav} \sigma_*$ denote that two states σ and σ_* agree on the auxiliary variables wrt. $cpav \subseteq \mathbf{AExp}^+$ in the sense that $\forall x \in \mathbf{Var}.\, \sigma(x) = \sigma_*(x)$ and $\forall a \in cpav.\, [\![a]\!]\sigma = \sigma_*(nv(a))$. We can then obtain the following soundness theorem.

Theorem 19 (Soundness of simple PRE)
If $s : ant, cpav \longrightarrow ant', cpav' \hookrightarrow s_*$ and $\sigma \sim_{cpav} \sigma_*$, then
— $\sigma \succ s \rightarrow \sigma'$ implies the existence of σ'_* such that $\sigma' \sim_{cpav'} \sigma'_*$ and $\sigma_* \succ s_* \rightarrow \sigma'_*$,
— $\sigma_* \succ s_* \rightarrow \sigma'_*$ implies the existence of σ' such that $\sigma' \sim_{cpav'} \sigma'_*$ and $\sigma \succ s \rightarrow \sigma'$.

Proof. We only prove the first half of the theorem. The proof of the other half is similar.

Given $s : ant, cpav \longrightarrow ant', cpav' \hookrightarrow s_*$, $\sigma \sim_{cpav} \sigma_*$ and $\sigma \succ s \rightarrow \sigma'$, we must find σ'_* such that $\sigma' \sim_{cpav'} \sigma'_*$ and $\sigma_* \succ s_* \rightarrow \sigma'_*$. The proof is by induction on the typing derivation with a subsidiary induction on the derivation of the semantic judgement. We look at the following nontrivial cases.

- Case $:=_{\mathrm{pre}}$: The type derivation is of the form

$$\frac{}{x := a : ant, cpav \longrightarrow ant', cpav' \hookrightarrow s_*}$$

where $ant =_{\mathrm{df}} ant' \setminus mod(x) \cup eval(a)$, $cpav' =_{\mathrm{df}} (cpav \cup eval(a) \setminus mod(x)) \cap ant'$. We note that from the constraint $cpav \subseteq ant$ it follows that $cpav \subseteq cpav' \cup$

$$\frac{a \notin cpav \quad a \notin ant' \vee x \in FV(a)}{x := a : ant' \setminus mod(x) \cup eval(a), cpav \longrightarrow ant', (cpav \cup eval(a) \setminus mod(x)) \cap ant' \hookrightarrow x := a} :=_{1\text{pre}}$$

$$\frac{a \notin cpav \quad a \in ant' \quad x \notin FV(a)}{\begin{array}{r} x := a : ant' \setminus mod(x) \cup eval(a), cpav \longrightarrow ant', (cpav \cup eval(a) \setminus mod(x)) \cap ant' \\ \hookrightarrow nv(a) := a; x := nv(a) \end{array}} :=_{2\text{pre}}$$

$$\frac{a \in cpav}{x := a : ant' \setminus mod(x) \cup eval(a), cpav \longrightarrow ant', (cpav \cup eval(a) \setminus mod(x)) \cap ant' \hookrightarrow x := nv(a)} :=_{3\text{pre}}$$

$$\frac{}{\mathsf{skip} : ant, cpav \longrightarrow ant, cpav \hookrightarrow \mathsf{skip}} \text{ skip}_{\text{pre}}$$

$$\frac{s_0 : ant, cpav \longrightarrow ant'', cpav'' \hookrightarrow s'_0 \quad s_1 : ant'', cpav'' \longrightarrow ant', cpav' \hookrightarrow s'_1}{s_0; s_1 : ant, cpav \longrightarrow ant', cpav' \hookrightarrow s'_0; s'_1} \text{ comp}_{\text{pre}}$$

$$\frac{s_t : ant, cpav \longrightarrow ant', cpav' \hookrightarrow s'_t \quad s_f : ant, cpav \longrightarrow ant', cpav' \hookrightarrow s'_f}{\mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f : ant, cpav \longrightarrow ant', cpav' \hookrightarrow \mathsf{if}\ b\ \mathsf{then}\ s'_t\ \mathsf{else}\ s'_f} \text{ if}_{\text{pre}}$$

$$\frac{s_t : ant, cpav \longrightarrow ant, cpav \hookrightarrow s'_t}{\mathsf{while}\ b\ \mathsf{do}\ s_t : ant, cpav \longrightarrow ant, cpav \hookrightarrow \mathsf{while}\ b\ \mathsf{do}\ s'_t} \text{ while}_{\text{pre}}$$

$$\frac{ant, cpav \leq ant_0, cpav_0 \quad s : ant_0, cpav_0 \longrightarrow ant'_0, cpav'_0 \hookrightarrow s' \quad ant'_0, cpav'_0 \leq ant', cpav'}{s : ant, cpav \longrightarrow ant', cpav' \hookrightarrow [\, nv(a) := a \mid a \in cpav_0 \setminus cpav\,]; s'; [\, nv(a) := a \mid a \in cpav' \setminus cpav'_0\,]} \text{ conseq}_{\text{pre}}$$

Figure 4.19: Type system for simple PRE, with the optimization component

$eval(a)$:

$$\begin{array}{rcl} cpav & = & cpav \cap ant \\ & = & cpav \cap ((ant' \setminus mod(x)) \cup eval(a)) \\ & \subseteq & (cpav \setminus mod(x) \cap ant') \cup eval(a) \\ & = & ((cpav \cup eval(a)) \setminus mod(x) \cap ant') \cup eval(a) \\ & = & cpav' \cup eval(a) \end{array}$$

and $cpav' \cap mod(x) = \emptyset$. The corresponding given semantic derivation must be of the form

$$\frac{}{\sigma \succ x := a \rightarrow \sigma[x \mapsto [\![a]\!]\sigma]}$$

hence $\sigma' = \sigma[x \mapsto [\![a]\!]\sigma]$.

– Subcase $:=_{1\text{pre}}$: We know that $a \notin cpav$. We also know that either $a \notin ant'$ or $x \in FV(a)$ (i.e., $a \notin mod(x)$), so $cpav \supseteq cpav'$. Moreover, $s_* =_{\text{df}} x := a$.

  We have the semantic derivation

$$\frac{}{\sigma_* \succ x := a \rightarrow \sigma'_*}$$

  where $\sigma'_* =_{\text{df}} \sigma_*[x \mapsto [\![a]\!]\sigma_*]$. From $\sigma \sim_{cpav} \sigma_*$ it follows that $[\![a]\!]\sigma = [\![a]\!]\sigma_*$, so that, using $cpav \supseteq cpav'$ as well, we can conclude $\sigma' = \sigma[x \mapsto [\![a]\!]\sigma] \sim_{cpav'} \sigma_*[x \mapsto [\![a]\!]\sigma] = \sigma_*[x \mapsto [\![a]\!]\sigma_*] = \sigma'_*$.

– Subcase $:=_{2\text{pre}}$: We have that $a \notin cpav$. We also have that a is nontrivial and $cpav \cup \{a\} \supseteq cpav'$. Also, $s_* =_{\text{df}} nv(a) := a; x := nv(a)$. We have the semantic derivation

$$\frac{\overline{\sigma_* \succ nv(a) := a \to \sigma''_*} \qquad \overline{\sigma''_* \succ x := nv(a) \to \sigma'_*}}{\sigma_* \succ nv(a) := a; x := nv(a) \to \sigma'_*.}$$

where $\sigma''_* =_{\text{df}} \sigma_*[nv(a) \mapsto [\![a]\!]\sigma_*]$ and $\sigma'_* =_{\text{df}} \sigma''_*[x \mapsto \sigma''_*(nv(a))] = \sigma''_*[x \mapsto [\![a]\!]\sigma_*]$. From $\sigma \sim_{cpav} \sigma_*$ it is immediate that $\sigma \sim_{cpav \cup \{a\}} \sigma_*[nv(a) \mapsto [\![a]\!]\sigma] = \sigma_*[nv(a) \mapsto [\![a]\!]\sigma_*] = \sigma''_*$ and therefore by $cpav \cup \{a\} \supseteq cpav'$ we have $\sigma' = \sigma[x \mapsto [\![a]\!]\sigma] \sim_{cpav'} \sigma''_*[x \mapsto [\![a]\!]\sigma] = \sigma''_*[x \mapsto [\![a]\!]\sigma_*] = \sigma'_*$.

– Subcase $:=_{3\text{pre}}$: We have that $a \in cpav$, so it follows that $cpav \supseteq cpav'$. We have $s_* =_{\text{df}} x := nv(a)$. We have the semantic derivation

$$\overline{\sigma_* \succ x := nv(a) \to \sigma'_*}$$

where $\sigma'_* =_{\text{df}} \sigma_*[x \mapsto \sigma_*(nv(a))]$. We know that $a \in cpav$, so from $\sigma \sim_{cpav} \sigma_*$ we learn $[\![a]\!]\sigma = \sigma_*(nv(a))$. Further, using also that $cpav \supseteq cpav'$, we realize that $\sigma' = \sigma[x \mapsto [\![a]\!]\sigma] \sim_{cpav'} \sigma_*[x \mapsto [\![a]\!]\sigma] = \sigma_*[x \mapsto \sigma_*(nv(a))] = \sigma'_*$.

- Case $\text{conseq}_{\text{pre}}$: The type derivation is of the form

$$\frac{\begin{array}{c}\vdots \\ s : ant_0, cpav_0 \longrightarrow ant'_0, cpav'_0 \hookrightarrow s_*\end{array}}{s : ant, cpav \longrightarrow ant', cpav' \hookrightarrow s'; s_*; s''}$$

where $(ant, cpav) \leq (ant_0, cpav_0)$, $(ant'_0, cpav'_0) \leq (ant', cpav')$, $s' =_{\text{df}} [nv(a) := a \mid a \in cpav_0 \setminus cpav]$ and $s'' =_{\text{df}} [nv(a) := a \mid a \in cpav' \setminus cpav'_0]$. First we find σ_0 such that $\sigma_* \succ s' \to \sigma_0$ and $\sigma \sim_{cpav_0} \sigma_0$. We have the semantic derivation

$$\overline{\sigma_* \succ s' \to \sigma_0}$$

where $\sigma_0 =_{\text{df}} \sigma_*[nv(a) \mapsto [\![a]\!]\sigma_* \mid a \in cpav_0 \setminus cpav]$. From $\sigma \sim_{cpav} \sigma_*$ we get that $\sigma \sim_{cpav_0} \sigma_*[nv(a) \mapsto [\![a]\!]\sigma \mid a \in cpav_0 \setminus cpav] = \sigma_*[nv(a) \mapsto [\![a]\!]\sigma_* \mid a \in cpav_0 \setminus cpav] = \sigma_0$, since every expression in the difference of $cpav_0$ and $cpav$ is explicitly made equal to its corresponding auxiliary variable and no variables from **Var** are modified. From the induction hypothesis we obtain that there is a state σ_1 such that $\sigma_0 \succ s_* \to \sigma_1$ and $\sigma' \sim_{cpav'_0} \sigma_1$. It is now enough to show that there is a state σ'_* such that $\sigma_1 \succ s'' \to \sigma'_*$ and $\sigma' \sim_{cpav'} \sigma'_*$. Similarly for the case of s', we have the derivation

$$\overline{\sigma_1 \succ s'' \to \sigma'_*}$$

where $\sigma'_* =_{\text{df}} \sigma_1[nv(a) \mapsto [\![a]\!]\sigma_1 \mid a \in cpav' \setminus cpav'_0]$. Again it is easy to realize that $\sigma' \sim_{cpav'_0} \sigma_1$ gives us $\sigma' \sim_{cpav'} \sigma_1[nv(a) \mapsto [\![a]\!]\sigma' \mid a \in cpav' \setminus cpav'_0] = \sigma_1[nv(a) \mapsto [\![a]\!]\sigma_1 \mid a \in cpav' \setminus cpav'_0] = \sigma'_*$.

$\square$

It is possible to show more than just correctness of the optimization using the relational method. One can also show that the optimization is actually an improvement in the sense that the number of evaluations of an expression on any given program path cannot increase. This means that no new computations can be introduced which are not used later on in the program. This is not obvious, since code motion might introduce unneeded evaluations.

To show this property, there must be a way to count expression uses. This can be done via a simple instrumented semantics, which counts the number of evaluations of every expression. In the instrumented semantics a state is a pair (σ, r) of a standard state $\sigma \in \mathbf{Var} \to \mathbb{Z}$ (an assignment of integer values to variables) and a "resource" state $r \in \mathbf{AExp}^+ \to \mathbb{N}$ associating to every nontrivial arithmetic expression a natural number for the number of times it has been evaluated. The rules of the semantics are as those for the standard semantics, except that for assignments of nontrivial expressions we stipulate

$$\frac{}{(\sigma, r) \succ x := a \to (\sigma[x \mapsto [\![a]\!]\sigma], r[a \mapsto r(a) + 1])}$$

The corresponding similarity relation between the states is the following. We define $(\sigma, r) \approx_{cpav} (\sigma_*, r_*)$ to mean that two states (σ, r) and (σ_*, r_*) are similar wrt. $cpav \subseteq \mathbf{AExp}^+$ in the sense that $\sigma \sim_{cpav} \sigma_*$ and, moreover, $\forall a \in cpav.\, r_*(a) \leq r(a) + 1$ and $\forall a \in \mathbf{AExp}^+ \setminus cpav.\, r_*(a) \leq r(a)$.

Here the conditional partial availability types serve as an "amortization" mechanism. The intuitive meaning of an expression being in the type of a program point is that there will be a use of this expression somewhere in the future, where this expression will be replaced with a variable already holding its value. Thus it is possible that a computation path of an optimized program has one more evaluation of the expression before this point than the corresponding computation path of the original program due to an application of subsumption. This does not break the improvement argument, since the type increase at the subsumption point contains a promise that this evaluation will be taken advantage of ("amortized") in the future.

Theorem 20 (Improvement property of simple PRE)
If $s : ant, cpav \longrightarrow ant', cpav' \hookrightarrow s_*$ *and* $(\sigma, r) \approx_{cpav} (\sigma_*, r_*)$, *then*
— $(\sigma, r) \succ s \to (\sigma', r')$ *implies the existence of* (σ'_*, r'_*) *such that* $(\sigma', r') \approx_{cpav'} (\sigma'_*, r'_*)$ *and* $(\sigma_*, r_*) \succ s_* \to (\sigma'_*, r'_*)$,
— $(\sigma_*, r_*) \succ s_* \to (\sigma'_*, r'_*)$ *implies the existence of* (σ', r') *such that* $(\sigma', r') \approx_{cpav'} (\sigma'_*, r'_*)$ *and* $(\sigma, r) \succ s \to (\sigma', r')$.

Proof. Again we only show the proof of the first part. The reasoning is similar to that we showed in the proof of Theorem 19, so we only concentrate ob the evaluation counting part of the states and ignore the variable values.

Given $s : ant, cpav \longrightarrow ant', cpav' \hookrightarrow s_*$, $(\sigma, r) \approx_{cpav} (\sigma_*, r_*)$ and $(\sigma, r) \succ s \to (\sigma', r')$, we look for a state (σ'_*, r'_*) such that $(\sigma', r') \approx_{cpav'} (\sigma'_*, r'_*)$ and $(\sigma_*, r_*) \succ s_* \to (\sigma'_*, r'_*)$.

The proof is by induction on the type derivation and we look at the following non-trivial cases.

- Case $:=_{\text{pre}}$: The type derivation is of the form

$$\overline{x := a : ant, cpav \longrightarrow ant', cpav' \hookrightarrow s_*}$$

where $ant =_{\text{df}} ant' \setminus mod(x) \cup eval(a)$, $cpav' =_{\text{df}} (cpav \cup eval(a) \setminus mod(x)) \cap ant'$.

From Theorem 19, we remember that $cpav \subseteq cpav' \cup eval(a)$.

At the same time also $cpav \cup eval(a) \supseteq (cpav \cup eval(a)) \setminus mod(x) \cap ant' = cpav'$. So for any nontrivial expression $a' \neq a$, $a' \in cpav$ and $a' \in cpav'$ are in fact equivalent statements.

The given semantic derivation must be of the form

$$\overline{(\sigma, r) \succ x := a \rightarrow (\sigma[x \mapsto [\![a]\!]\sigma], r[a \mapsto r(a) + 1])}$$

  - Subcase $:=_{1\text{pre}}$: We have that $a \notin cpav$ and either $a \notin ant'$ or $x \in FV(a)$, so $a \notin cpav'$. Moreover, $s_* =_{\text{df}} x := a$.

    We have the semantic derivation

    $$\overline{(\sigma_*, r_*) \succ x := a \rightarrow (\sigma'_*, r'_*)}\ .$$

    where $(\sigma'_*, r'_*) =_{\text{df}} (\sigma_*[x \mapsto [\![a]\!]\sigma_*], r_*[a \mapsto r_*(a) + 1])$. From the assumption we know that $r_*(a) \leq r(a)$, so $r'_*(a) = r_*(a) + 1 \leq r(a) + 1 = r'(a)$.

  - Subcase $:=_{2\text{pre}}$: We have that $a \notin cpav$ and both $a \in ant'$ and $x \notin FV(a)$, hence $a \in cpav'$. Moreover, $s_* =_{\text{df}} nv(a) := a; x := nv(a)$. We have the semantic derivation

    $$\frac{\overline{(\sigma_*, r_*) \succ nv(a) := a \rightarrow \sigma''_*, r''_*} \qquad \overline{(\sigma''_*, r''_*) \succ x := nv(a) \rightarrow (\sigma'_*, r'_*)}}{(\sigma_*, r_*) \succ nv(a) := a; x := nv(a) \rightarrow (\sigma'_*, r'_*)}\ .$$

    where $(\sigma''_*, r''_*) =_{\text{df}} (\sigma_*[nv(a) \mapsto [\![a]\!]\sigma_*], r_*[a \mapsto r_*(a)+1])$ and $(\sigma'_*, r'_*) =_{\text{df}} (\sigma''_*[x \mapsto \sigma''_*(nv(a))], r''_*)$. Similarly to the previous case, from $r_*(a) \leq r(a)$ we obtain $r'_*(a) = r''_*(a) = r_*(a) + 1 \leq r(a) + 1 = r'(a) < r'(a) + 1$.

  - Subcase $:=_{3\text{pre}}$: We have that $a \in cpav$ and $s_* =_{\text{df}} x := nv(a)$. We have the derivation

    $$\overline{(\sigma_*, r_*) \succ x := nv(a) \rightarrow (\sigma'_*, r'_*)}\ .$$

    where $(\sigma'_*, r'_*) =_{\text{df}} (\sigma_*[x \mapsto \sigma_*(nv(a))], r_*)$. From $r_*(a) \leq r(a) + 1$ we get that $r'_*(a) = r_*(a) \leq r(a) + 1 = r'(a)$ (so all is well both if $a \notin cpav'$ and if $a \in cpav'$; both situations are possible).

  In all three subcases, for any nontrivial $a' \neq a$, if $a' \in cpav$, then we have that $a' \in cpav'$ as well and therefore from $r_*(a') \leq r(a') + 1$ we get $r'_*(a') = r_*(a') \leq r(a') + 1 = r'(a') + 1$. Similarly, for $a' \neq a$ such that $a' \notin cpav$ we have $a' \notin cpav'$, so $r_*(a') \leq r(a')$ gives us $r'_*(a') = r_*(a') \leq r(a') = r'(a')$.

- Case conseq$_{\text{pre}}$: The type derivation is of the form

$$\frac{\begin{array}{c}\vdots\\ s : ant_0, cpav_0 \longrightarrow ant_0', cpav_0' \hookrightarrow s_*\end{array}}{s : ant, cpav \longrightarrow ant', cpav' \hookrightarrow s'; s_*; s''}$$

  where $(ant, cpav) \leq (ant_0, cpav_0)$, $(ant_0', cpav_0') \leq (ant', cpav')$ and $s' =_{\text{df}} [nv(a) := a \mid a \in cpav_0 \setminus cpav]$, $s'' =_{\text{df}} [nv(a) := a \mid a \in cpav' \setminus cpav_0']$. First we find a state (σ_0, r_0) such that $(\sigma_*, r_*) \succ s' \rightarrow (\sigma_0, r_0)$ and $(\sigma, r) \approx_{cpav_0} (\sigma_0, r_0)$. We have the semantic derivation

$$\overline{(\sigma_*, r_*) \succ s' \rightarrow (\sigma_0, r_0)}\ .$$

  where $(\sigma_0, r_0) =_{\text{df}} (\sigma_*[nv(a) \mapsto [\![a]\!]\sigma_* \mid a \in cpav_0 \setminus cpav], r_*[a \mapsto r_*(a)+1 \mid a \in cpav_0 \setminus cpav])$. For any expression $a \in cpav$, from $cpav \subseteq cpav_0$ we have $a \in cpav_0$, whereas from $r_*(a) \leq r(a)+1$ we can conclude $r_0(a) = r_*(a) \leq r(a)+1$. Similarly, for any nontrivial expression $a \notin cpav_0$, from $cpav \subseteq cpav_0$ we learn that $a \notin cpav$, and then $r_*(a) \leq r(a)$ tells us that $r_0(a) = r_*(a) \leq r(a)$. If an expression a is in $cpav_0 \setminus cpav$, then from $r_*(a) \leq r(a)$ we can conclude $r_0(a) = r_*(a) + 1 \leq r(a) + 1$.

  By the induction hypothesis, there must exist a state (σ_1, r_1) such that $(\sigma_0, r_0) \succ s_* \rightarrow (\sigma_1, r_1)$ and $(\sigma', r') \approx_{cpav_0'} (\sigma_1, r_1)$. It is now enough to exhibit a state (σ_*', r_*') such that $(\sigma_1, r_1) \succ s'' \rightarrow (\sigma_*', r_*')$ and $(\sigma', r') \approx_{cpav'} (\sigma_*', r_*')$. This can be done in the same way as for s'.

□

To prove that an optimization is really optimal in the sense of achieving the best possible improvement (which simple PRE really is not), we would have to fix what kind of modifications of a given program we consider as possible transformation candidates (they should not modify the control flow graph other than by splitting edges, they should not take advantage of the real domains and interpretation of expressions etc.). The argument would have to compare the optimization to other sound transformation candidates.

For proof transformation, the key observation is that the expressions which are conditionally partially available must have been computed and their values not modified, thus their values are equal to the values of the corresponding auxiliary variables that have been defined.

Let $P|_{cpav}$ abbreviate $\bigwedge[\, nv(a) = a \mid a \in cpav\,] \wedge P$. We have the following theorem.

Theorem 21 (Preservation of Hoare logic provability/proofs)
If $s : ant, cpav \longrightarrow ant', cpav' \hookrightarrow s_*$*, then* $\{P\}\, s\, \{Q\}$ *implies* $\{P|_{cpav}\}\, s_*\, \{Q|_{cpav'}\}$.

Proof. Nonconstructively, this theorem is a corollary from the correctness of the optimization (second half) and soundness and relative completeness of Hoare logic.

We present a constructive proof which yields automatic Hoare proof transformation. Given a derivation of $s : ant, cpav \longrightarrow ant', cpav' \hookrightarrow s_*$ and an aligned Hoare proof of $\{P\}\, s\, \{Q\}$, we induct on the type derivation and transform the given Hoare proof into one of $\{P|_{cpav}\}\, s_*\, \{Q|_{cpav'}\}$.

We look at the cases where actual modifications happen (the cases for sequence, if, and while constructs are again straightforward).

- Case $:=_{\text{pre}}$: The type derivation is

$$\frac{}{x := a : ant, cpav \longrightarrow ant', cpav' \hookrightarrow s_*}$$

  where $ant =_{\text{df}} ant' \setminus mod(x) \cup eval(a)$, $cpav' =_{\text{df}} (cpav \cup eval(a) \setminus mod(x)) \cap ant'$. We notice that this implies $cpav \cup eval(a) \supseteq cpav'$ and $cpav' \cap mod(x) = \emptyset$. The latter observation gives that $P[a'/x]|_{cpav'} \Leftrightarrow P|_{cpav'}[a'/x]$.

  The given Hoare logic proof is

$$\frac{}{\{P[a/x]\}\, x := a\, \{P\}}$$

  - Subcase $:=_{1\text{pre}}$: We have that $a \notin cpav$. We also have that either $a \notin ant'$ or $x \in FV(a)$ (i.e., $a \notin mod(x)$), so $cpav \supseteq cpav'$. Moreover, $s_* =_{\text{df}} x := a$.

    From $cpav \supseteq cpav'$ it follows that $P[a/x]|_{cpav} \models P[a/x]|_{cpav'}$. The transformed Hoare logic proof is

$$\cfrac{\cfrac{\cfrac{}{\{P|_{cpav'}[a/x]\}\, x := a\, \{P|_{cpav'}\}}}{\{P[a/x]|_{cpav'}\}\, x := a\, \{P|_{cpav'}\}}}{\{P[a/x]|_{cpav}\}\, x := a\, \{P|_{cpav'}\}}$$

  - Subcase $:=_{2\text{pre}}$: We have that $a \notin cpav$. We also have that a is nontrivial, so that $cpav \cup \{a\} \supseteq cpav'$. Moreover, $s_* =_{\text{df}} nv(a) := a; x := nv(a)$.

    From $cpav \cup \{a\} \supseteq cpav'$ it follows that $P[nv(a)/x]|_{cpav \cup \{a\}} \models P[nv(a)/x]|_{cpav'}$. From reflexivity of equality, $P[a/x]|_{cpav} \Leftrightarrow P[a/x]|_{cpav} \wedge a = a \Leftrightarrow (P[nv(a)/x]|_{cpav \cup \{a\}})[a/nv(a)]$. The transformed Hoare logic proof is

$$\frac{B_0 \quad B_1}{\{P[a/x]|_{cpav}\}\, nv(a) = a; x := nv(a)\, \{P|_{cpav'}\}}$$

    where $B_0 \equiv$

$$\cfrac{\cfrac{}{\{P[nv(a)/x]|_{cpav \cup \{a\}}[a/nv(a)]\}\, nv(a) = a\, \{P[nv(a)/x]|_{cpav \cup \{a\}}\}}}{\{P[a/x]|_{cpav}\}\, nv(a) = a\, \{P[nv(a)/x]|_{cpav'}\}}$$

and $B_1 \equiv$

$$\dfrac{\overline{\{P|_{cpav'}[nv(a)/x]\}\, x := nv(a)\, \{P|_{cpav'}\}}}{\{P[nv(a)/x]|_{cpav'}\}\, x := nv(a)\, \{P|_{cpav'}\}}$$

– Subcase $:=_{3\text{pre}}$: We have that $a \in cpav$, so $cpav \subseteq cpav'$. Moreover, $s_* =_{\text{df}} x := nv(a)$.

  We have $a \in cpav$ and $cpav \supseteq cpav'$, therefore $P[a/x]|_{cpav} \models P[a/x]|_{cpav'} \wedge nv(a) = a$. Substitution of equals for equals gives $P|_{cpav'}[a/x] \wedge nv(a) = a \models P|_{cpav'}[nv(a)/x]$. The transformed Hoare logic proof is

$$\dfrac{\dfrac{\dfrac{\overline{\{P|_{cpav'}[nv(a)/x]\}\, x := nv(a)\, \{P|_{cpav'}\}}}{\{P|_{cpav'}[a/x] \wedge nv(a) = a\}\, x := nv(a)\, \{P|_{cpav'}\}}}{\{P[a/x]|_{cpav'} \wedge nv(a) = a\}\, x := nv(a)\, \{P|_{cpav'}\}}}{\{P[a/x]|_{cpav}\}\, x := nv(a)\, \{P|_{cpav'}\}}$$

- Case $\text{conseq}_{\text{pre}}$: The type derivation is

$$\dfrac{\begin{array}{c}\vdots\\ s : ant_0, cpav_0 \longrightarrow ant'_0, cpav'_0 \hookrightarrow s_*\end{array}}{s : ant, cpav \longrightarrow ant', cpav' \hookrightarrow s'; s_*; s''}$$

  where $(ant, cpav) \leq (ant_0, cpav_0)$, $(ant'_0, cpav'_0) \leq (ant, cpav)$ and $s' =_{\text{df}} [nv(a) := a \mid a \in cpav_0 \setminus cpav]$, $s'' =_{\text{df}} [nv(a) := a \mid a \in cpav' \setminus cpav'_0]$. The given Hoare logic proof is

$$\dfrac{\begin{array}{c}\vdots\\ \{P_0\}\, s\, \{Q_0\}\end{array}}{\{P\}\, s\, \{Q\}}$$

  where $P \models P_0$ and $Q_0 \models Q$.

  By the induction hypothesis, there is a Hoare logic proof of $\{P_0|_{cpav_0}\}\, s_*\, \{Q_0|_{cpav'_0}\}$. It is an assumption that $P \models P_0$, hence $P|_{cpav} \models P_0|_{cpav}$.

  By reflexivity of equality $P_0|_{cpav} \Leftrightarrow P_0|_{cpav} \wedge \bigwedge[a = a| \mid a \in cpav_0 \setminus cpav\,] \models P_0|_{cpav_0}[\,a/nv(a) \mid a \in cpav_0 \setminus cpav\,]$. Hence from the axiom $\{P_0|_{cpav_0}[\,a/nv(a) \mid a \in cpav_0 \setminus cpav\,]\}\, s'\, \{P_0|_{cpav_0}\}$ by the consequence rule we have a proof of $\{P|_{cpav}\}\, s'\, \{P_0|_{cpav_0}\}$.

  Similarly we can make a proof of $\{Q_0|_{cpav'_0}\}\, s''\, \{Q|_{cpav'}\}$.

  Putting everything together with the sequence rule, we obtain a proof of $\{P|_{cpav}\}\, s'; s_*; s''\, \{Q|_{cpav'}\}$, which is the required transformed Hoare logic proof.

□

An example application of the type system and transformation of Hoare logic proofs is shown in Figures 4.20, 4.21 and 4.22. We have a program $s =_{\text{df}}$ while $i <$

$$n := n + (y+z) : \{y+z\}, \{y+z\} \longrightarrow \{y+z\}, \{y+z\} \hookrightarrow n := n + t$$

$$x := y + z : \{y+z\}, \{y+z\} \longrightarrow \emptyset, \emptyset \hookrightarrow x := t$$

$$i := i + 1 : \{y+z\}, \{y+z\} \longrightarrow \{y+z\}, \{y+z\} \hookrightarrow i := i + 1$$

$$n := n + (y+z); i := i + 1 : \{y+z\}, \{y+z\} \longrightarrow \{y+z\}, \{y+z\} \hookrightarrow n := n + t; i := i + 1$$

$$\text{while } i < k \text{ do } (n := n + (y+z); i := i + 1) : \{y+z\}, \{y+z\} \longrightarrow \{y+z\}, \{y+z\} \hookrightarrow \text{while } i < k \text{ do } (n := n + t; i := i + 1)$$

$$\text{while } i < k \text{ do } (n := n + (y+z); i := i + 1) : \{y+z\}, \emptyset \longrightarrow \{y+z\}, \{y+z\} \hookrightarrow t := y + z; \text{while } i < k \text{ do } (n := n + t; i := i + 1)$$

$$\text{while } i < k \text{ do } (n := n + (y+z); i := i + 1); x := y + z : \{y+z\}, \emptyset \longrightarrow \emptyset, \emptyset \hookrightarrow t := y + z; \text{while } i < k \text{ do } (n := n + t; i := i + 1); x := t$$

Figure 4.20: Type derivation for the example program

$$\{n = k * (y+z)\}\ x := y + z\ \{n = k * (y+z)\}$$

$$\{i + 1 \leq k \wedge n = (i+1) * (y+z)\}\ i := i + 1\ \{i \leq k \wedge n = i * (y+z)\}$$

$$\{i + 1 \leq k \wedge n + (y+z) = (i+1) * (y+z)\}\ n := n + (y+z)\ \{i + 1 \leq k \wedge n = (i+1) * (y+z)\}$$

$$\{i < k \wedge n = i * (y+z)\}\ n := n + (y+z); i := i + 1\ \{i \leq k \wedge n = i * (y+z)\}$$

$$\{i \leq k \wedge n = i * (y+z)\}\ \text{while } i < k \text{ do } (n := n + (y+z); i := i + 1)\ \{i \not< k \wedge i \leq k \wedge n = i * (y+z)\}$$

$$\{n = 0 \wedge i = 0 \wedge k \geq 0\}\ \text{while } i < k \text{ do } (n := n + (y+z); i := i + 1); x := y + z\ \{n = k * (y+z)\}$$

Figure 4.21: The original proof for the example program

$$\{i + 1 \leq k \wedge n = (i+1) * (y+z) \wedge t = y + z\}\ i := i + 1\ \{i \leq k \wedge n = i * (y+z) \wedge t = y + z\}$$

$$\{i \leq k \wedge n = i * (y+z)\}\ t := y + z\ \{i \leq k \wedge n = i * (y+z) \wedge t = y + z\}$$

$$\{i + 1 \leq k \wedge n + t = (i+1) * (y+z) \wedge t = y + z\}\ n := n + t\ \{i + 1 \leq k \wedge n = (i+1) * (y+z) \wedge t = y + z\}$$

$$\{i < k \wedge n = i * (y+z) \wedge t = y + z\}\ n := n + t; i := i + 1\ \{i \leq k \wedge n = i * (y+z) \wedge t = y + z\}$$

$$\{i \leq k \wedge n = i * (y+z) \wedge t = y + z\}\ \text{while } i < k \text{ do } (n := n + (y+z); i := i + 1)\ \{i \not< k \wedge i \leq k \wedge n = i * (y+z) \wedge t = y + z\}$$

$$\{n = k * (y+z)\}\ x := t\ \{n = k * (y+z)\}$$

$$\{i \leq k \wedge n = i * (y+z)\}\ t := y + z; \text{while } i < k \text{ do } (n := n + (y+z); i := i + 1)\ \{i = k \wedge n = i * (y+z) \wedge t = y + z\}$$

$$\{n = k * (y+z) \wedge t = y + z\}\ x := t\ \{n = k * (y+z)\}$$

$$\{n = 0 \wedge i = 0 \wedge k \geq 0\}\ t := y + z; \text{while } i < k \text{ do } (n := n + t; i := i + 1); x := t\ \{n = k * (y+z)\}$$

Figure 4.22: The transformed proof

$$\{i+1 \leq k \wedge c+1 = i+1\}\, n := n + (y+z)\, \{i+1 \leq k \wedge c = (i+1)\}$$

$$\{c+1 = k+1\}\, x := y+z\, \{c = k+1\}$$

$$\{i+1 \leq k \wedge c = (i+1)\}\, i := i+1\, \{i \leq k \wedge c = i\}$$

$$\{i < k \wedge c = i\}\, n := n + (y+z); i := i+1\, \{i \leq k \wedge c = i)\}$$

$$\{i \leq k \wedge c = i\}\, \mathsf{while}\ i < k\ \mathsf{do}\ (n := n + (y+z); i := i+1)\, \{i \not< k \wedge i \leq k \wedge c = i)\}$$

$$\{c = 0 \wedge i = 0 \wedge k \geq 0\}\, \mathsf{while}\ i < k\ \mathsf{do}\ (n := n + (y+z); i := i+1); x := y+z\, \{c = k+1)\}$$

Figure 4.23: An original proof for resource usage

$$\{i \leq k \wedge c+1 \leq i+1\}\, t := y+z\, \{i \leq k \wedge c \leq i+1\}$$

$$\{c \leq k+1\}\, x := t\, \{c \leq k+1)\}$$

$$\{i+1 \leq k \wedge c \leq (i+1)+1\}\, i := i+1\, \{i \leq k \wedge c \leq i+1\}$$

$$\{i+1 \leq k \wedge c \leq (i+1)+1\}\, n := n+t\, \{i+1 \leq k \wedge c \leq (i+1)+1\}$$

$$\{i < k \wedge c \leq i+1\}\, n := n+t; i := i+1\, \{i \leq k \wedge c \leq i+1\}$$

$$\{i \leq k \wedge c \leq i+1\}\, \mathsf{while}\ i < k\ \mathsf{do}\ (n := n + (y+z); i := i+1)\, \{i \not< k \wedge i \leq k \wedge c \leq i+1\}$$

$$\{i \leq k \wedge c \leq i\}\, t := y+z; \mathsf{while}\ i < k\ \mathsf{do}\ (n := n + (y+z); i := i+1)\, \{i = k \wedge c \leq i+1\}$$

$$\{c \leq 0 \wedge i = 0 \wedge k \geq 0\}\, t := y+z; \mathsf{while}\ i < k\ \mathsf{do}\ (n := n+t; i := i+1); x := t\, \{c \leq k+1\}$$

Figure 4.24: The transformed proof for resource usage

k do $(n := n + (y+z); i := i+1); x := y+z$ and a Hoare derivation tree for $\{n = 0 \wedge i = 0 \wedge k \geq 0\}\, s\, \{n = k * (y+z)\}$. Note that to make the derivation trees smaller, we use $n := n + (y+z)$, i.e. an expression with more than one operator. This can be considered as syntactic sugar, since the assignment could be rewritten as $n' := y+z; n := n+n'$. The optimization lifts the computation of $y+z$ out of the while-loop. This renders the original proof of the program impossible to associate to the transformed program. For example, the old loop invariant is not valid any more, since it talks about $y+z$, but the expression is not present in the modified loop. Figure 4.22 shows the proof tree where this has been remedied using the information present in the types.

We can also achieve an automatic proof transformation corresponding to the improvement property. This allows us to invoke a performance bound of a given program to obtain one for its optimized version.

Similarly to semantic improvement, where we needed an instrumented semantics, now we need an instrumented Hoare logic. We extend the signature of the standard Hoare logic with an extralogical constant $\ulcorner a \urcorner$ for any expression $a \in \mathbf{AExp}^+$. The inference rules of the instrumented Hoare logic are the analogous to those for the standard Hoare logic except that the axiom for nontrivial assignment becomes

$$\overline{\{P[a/x][\ulcorner a \urcorner + 1/\ulcorner a \urcorner]\}\, x := a\, \{P\}}$$

It should not come as a surprise that the instrumented Hoare logic is sound and

relatively complete wrt. the instrumented semantics.

Now, we define $P\|_{cpav}$ to abbreviate

$$\begin{array}{ll} [\,\exists v(a) \mid a \in \mathbf{AExp}^+\,]. & \\ & \bigwedge[\,nv(a) = a \wedge \ulcorner a \urcorner \leq v(a) + 1 \mid a \in cpav\,] \\ \wedge & \bigwedge[\,\ulcorner a \urcorner \leq v(a) \mid a \notin cpav\,] \\ \wedge & P[v(a)/\ulcorner a \urcorner]. \end{array}$$

Here $v(a)$ generates a new unique logic variable for every nontrivial arithmetic expression. With this notation we can state a refined theorem, yielding transformation of proofs of the instrumented Hoare logic.

Theorem 22 (Preservation of instrumented Hoare logic provability/proofs)
If $s : ant, cpav \longrightarrow ant', cpav' \hookrightarrow s_*$, *then* $\{P\}\, s\, \{Q\}$ *implies* $\{P\|_{cpav}\}\, s_*\, \{Q\|_{cpav'}\}$.

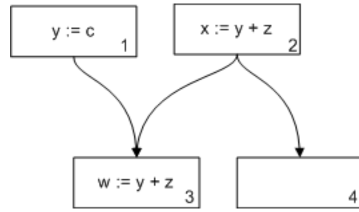
The proofs (nonconstructive and constructive) are similar to those of the previous theorem.

To witness the theorem in action we revisit the program analyzed in Figure 4.20. Figure 4.23 demonstrates that in the instrumented Hoare logic we can prove that the program computes $y + z$ exactly $k + 1$ times (we have abbreviated $\ulcorner y + z \urcorner$ to c). The invariant for the while-loop is $i \leq k \wedge c = i$. Figure 4.24 contains the transformed proof for the optimized program. We can prove that $y + z$ is computed at most $k + 1$ times, but the proof is quite different; in particular, the loop invariant is now $i \leq k \wedge c = i + 1$. (In this proof, we have enhanced readability by replacing the existentially quantified assertions yielded by the automatic transformation with equivalent quantifier-free simplifications.)

As expected, this formal counterpart of the semantic improvement argument is no smarter than the semantic improvement argument itself. In our semantic improvement statement we claimed that the optimized program performs at most one extra evaluation per expression as compared to the original program, namely for the expression that is precomputed for future use. Had we claimed something more specific and stronger about, e.g., those loops from where at least one assignment can be moved out, our corresponding automatic proof transformation could have been stronger as well. It is not our goal to delve deeper into this interesting point here. Rather, we are content here with the observation that constructive and structured semantic arguments have can be given formal counterparts in the form of automatic proof transformations.

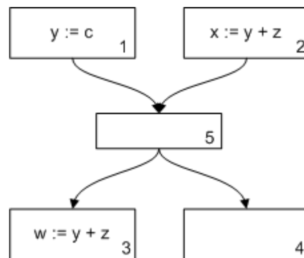
4.4.2 Full PRE

We now look at the formulation of full PRE by Paleri et al. [50]. As was explained in Section 4.4.1, simple PRE does not use all optimization opportunities. This stems from the fact that it only takes into account total anticipability. An example of a program which simple PRE does not optimize is the following one.



The program is left unoptimized by simple PRE since $y + z$ is not anticipable at the exit of node 2. Full PRE would optimize the program by introducing a new auxiliary variable to hold the computation of $y + z$ in node 2 and copying the computation of $y + z$ into the edge leaving node 1. This would allow to skip the computation of $y + z$ in node 3.

This does not mean that it is possible to simply substitute the total anticipability analysis with partial anticipability. The following example illustrates this.



While it is seemingly similar to the previous example, it cannot be optimized the same way, since if we moved a computation of $y + z$ into the edge (1,5), we would potentially worsen the runtime behavior of the program, since going through the program through nodes (1, 5, 4), there would be an extra evaluation of $y + z$ that was not present in the original program. In fact no further optimization of this program is possible.

The fundamental observation which allows us to perform PRE fully and correctly is that partial anticipability is enough only if the path leading from the node where the expression becomes available (node 2 in the examples) to a node where the expression becomes anticipable (node 3 in the examples) contains no nodes at which the expression is neither anticipable nor available.

The last condition can be detected by two additional dataflow analyses, thus the full PRE algorithm requires four analyses in total. These are standard (total) availability and anticipability, and safe partial availability and safe partial anticipability analyses. The two latter depend on availability and anticipability. Their descriptions rely on the notion of safety. A program point is said to be safe wrt. an expression if that expression is either available or anticipable at that program point.

The dataflow inequations for the whole program in the CFG representation are the following.

$$
\begin{array}{rcl}
ANTOUT_i & \subseteq & \begin{cases} \emptyset & \text{if } i = f \\ \bigcap_{j \in succ(i)} ANTIN_j & \text{otherwise} \end{cases} \\
ANTIN_i & \subseteq & ANTOUT_i \setminus MOD_i \cup EVAL_i \\
AVIN_i & \subseteq & \begin{cases} \emptyset & \text{if } i = s \\ \bigcap_{j \in pred(i)} AVOUT_i & \text{otherwise} \end{cases} \\
AVOUT_i & \subseteq & (AVIN_i \cup EVAL_i) \setminus MOD_i \\
SPANTOUT_i & \supseteq & \begin{cases} \emptyset & \text{if } i = f \\ \bigcup_{j \in succ(i)} SPANTIN_j & \text{otherwise} \end{cases} \\
SPANTIN_i & \supseteq & (SPANTOUT_i \setminus MOD_i \cup EVAL_i) \cap SAFEIN_i \\
SPAVIN_i & \supseteq & \begin{cases} \emptyset & \text{if } i = s \\ \bigcup_{j \in pred(i)} SPAVOUT_i & \text{otherwise} \end{cases} \\
SPAVOUT_i & \supseteq & ((SPAVIN_i \cup EVAL_i) \setminus MOD_i) \cap SAFEOUT_i
\end{array}
$$

$$
\begin{array}{rcl}
ANTOUT_i & \subseteq & SPANTOUT_i \subseteq SAFEOUT_i \\
ANTIN_i & \subseteq & SPANTIN_i \subseteq SAFEIN_i \\
AVIN_i & \subseteq & SPAVIN_i \subseteq SAFEIN_i \\
AVOUT_i & \subseteq & SPAVOUT_i \subseteq SAFEOUT_i \\
SAFEIN_i & = & ANTIN_i \cup AVIN_i \\
SAFEOUT_i & = & ANTOUT_i \cup AVOUT_i
\end{array}
$$

Using the results of the analysis, it is possible to optimize the program in the following way. A computation of an expression should be added to the edge (i, j), if the expression is safely partially available at the entry of node j, but not at the exit of node i. Furthermore, the expression should be safely partially anticipable at the entry of node j. This transformation makes partially redundant expressions fully redundant exactly in places where it is necessary, thus the checking of safe partial anticipability. Note that the latter was not necessary in simple PRE, since conditional partial availability already implied anticipability. In a node where an expression is evaluated if the expression is already safely partially available, its evaluation can be replaced with a use of the auxiliary variable. If the expression is not available, but is safely partially anticipable, the result of the evaluation can be saved in the auxiliary variable.

We now present these analyses and the optimization as type systems. The type system for anticipability was already described in Section 4.4.1. The type system for availability was shown in Section 4.3

The type systems for safe partial availability and safe partial anticipability are given as a single type system in Figure 4.25. They do not depend on each other, but depend on the safety component. We use $\underline{s}$ to denote a full type derivation of

$$\overline{\underline{x := a} : (spant' \backslash mod(x) \cup eval(a)) \cap safe, spav \longrightarrow spant', (spav \cup eval(a) \setminus mod(x)) \cap safe'}$$

$$\overline{\underline{\mathsf{skip}} : spant, spav \longrightarrow spant, spav}$$

$$\frac{\underline{s_0} : spant, spav \longrightarrow spant'', spav'' \quad \underline{s_1} : spant'', spav'' \longrightarrow spant', spav'}{\underline{s_0; s_1} : spant, spav \longrightarrow spant', spav'}$$

$$\frac{\underline{s_t} : spant, spav \longrightarrow spant', spav' \quad \underline{s_f} : spant, spav \longrightarrow spant', spav'}{\underline{\mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f} : spant, spav \longrightarrow spant', spav'}$$

$$\frac{\underline{s_t} : spant, spav \longrightarrow spant, spav}{\underline{\mathsf{while}\ b\ \mathsf{do}\ s_t} : spant, spav \longrightarrow spant, spav}$$

$$\frac{spant, spav \leq spant_0, spav_0 \quad \underline{s} : spant_0, spav_0 \longrightarrow spant'_0, spav'_0 \quad spant'_0, spav'_0 \leq spant', spav'}{\underline{s} : spant, spav \longrightarrow spant', spav'}$$

Figure 4.25: Type system for the underlying analyses of full PRE

$s : ant, av \longrightarrow ant', av'$, thus safety *safe* in the pretype of $\underline{s}$ is defined as $ant \cup av$, safety in the posttype $safe'$ is $ant' \cup av'$.

The complete type of a program point is thus $(ant, av, spant, spav) \in \mathcal{P}(\mathbf{AExp}^+) \times \mathcal{P}(\mathbf{AExp}^+) \times \mathcal{P}(\mathbf{AExp}^+) \times \mathcal{P}(\mathbf{AExp}^+)$, satisfying the conditions $ant \subseteq spant \subseteq ant \cup av$ and $av \subseteq spav \subseteq ant \cup av$. Subtyping for safe partial anticipability is reversed set inclusion, i.e., $\leq =_{\mathrm{df}} \supseteq$. For safe partial availability it is set inclusion, $\leq =_{\mathrm{df}} \subseteq$.

Without the extra restrictions on the types, the type system would still be sound, but it would lose its improvement property, i.e., it might allow optimizations which introduce unneeded evaluations. The restriction $ant \subseteq spant$ guarantees that the set of *fully* anticipable expression cannot be bigger than the set of *partially* anticipable expressions. This is guaranteed by the principal type inference algorithm, but in the type system the subsumption rule could break this relation without the extra restriction. The same holds for full and partial availability. The restrictions $spant \subseteq ant \cup av$ and $spav \subseteq ant \cup av$ guarantee *safety* as in the original algorithm of Paleri et al.

The optimizing type system for full PRE is given in Figure 4.26. Code motion at subsumption is now guided by the intersection of safe partial availability and safe partial anticipability. In the definition of soundness, the similarity relation on states also has to be invoked at this intersection. The same holds for proof transformation.

Theorem 23 (Soundness of full PRE)
If $\underline{s} : spant, spav \longrightarrow spant', spav' \hookrightarrow s_*$ *and* $\sigma \sim_{spant \cap spav} \sigma_*$*, then*
— $\sigma \succ s \rightarrow \sigma'$ *implies the existence of* σ'_* *such that* $\sigma' \sim_{spant' \cap spav'} \sigma'_*$ *and* $\sigma_* \succ s_* \rightarrow \sigma'_*$,
— $\sigma_* \succ s_* \rightarrow \sigma'_*$ *implies the existence of* σ' *such that* $\sigma' \sim_{spant' \cap spav'} \sigma'_*$ *and* $\sigma \succ s \rightarrow \sigma'$.

The proof is quite similar to that for simple PRE. Full PRE is using partial anticipability instead of full anticipability, but this does not affect the soundness

$$\frac{a \notin spav \quad a \notin spant \vee x \in FV(a)}{\underline{x := a} : (spant' \backslash mod(x) \cup eval(a)) \cap safe, spav \longrightarrow spant', (spav \cup eval(a) \setminus mod(x)) \cap safe' \hookrightarrow x := a}$$

$$\frac{a \notin spav \quad a \in spant \quad x \notin FV(a)}{\underline{x := a} : (spant' \backslash mod(x) \cup eval(a)) \cap safe, spav \longrightarrow spant', (spav \cup eval(a) \setminus mod(x)) \cap safe' \hookrightarrow nv(a) := a; x := nv(a)}$$

$$\frac{a \in spav}{\underline{x := a} : (spant' \backslash mod(x) \cup eval(a)) \cap safe, spav \longrightarrow spant', (spav \cup eval(a) \setminus mod(x)) \cap safe' \hookrightarrow x := nv(a)}$$

$$\overline{\underline{\mathsf{skip}} : spant, spav \longrightarrow spant, spav \hookrightarrow \mathsf{skip}}$$

$$\frac{\underline{s_0} : spant, spav \longrightarrow spant'', spav'' \hookrightarrow s_0' \quad \underline{s_1} : spant'', spav'' \longrightarrow spant', spav' \hookrightarrow s_1'}{\underline{s_0; s_1} : spant, spav \longrightarrow spant', spav' \hookrightarrow s_0'; s_1'}$$

$$\frac{\underline{s_t} : spant, spav \longrightarrow spant', spav' \hookrightarrow s_t' \quad \underline{s_f} : spant, spav \longrightarrow spant', spav' \hookrightarrow s_f'}{\underline{\mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f} : spant, spav \longrightarrow spant', spav' \hookrightarrow \mathsf{if}\ b\ \mathsf{then}\ s_t'\ \mathsf{else}\ s_f'}$$

$$\frac{\underline{s_t} : spant, spav \longrightarrow spant, spav \hookrightarrow s_t'}{\underline{\mathsf{while}\ b\ \mathsf{do}\ s_t} : spant, spav \longrightarrow spant, spav \hookrightarrow \mathsf{while}\ b\ \mathsf{do}\ s_t'}$$

$$\frac{spant, spav \leq spant_0, spav_0 \quad \underline{s} : spant_0, spav_0 \longrightarrow spant_0', spav_0' \hookrightarrow s' \quad spant_0', spav_0' \leq spant', spav'}{\underline{s} : spant, spav \longrightarrow spant', spav' \hookrightarrow [\, nv(a) := a \mid a \in (spav_0 \cap spant_0) \setminus spav \,]; s'; [\, nv(a) := a \mid a \in (spav' \cap spant') \setminus spav_0' \,]}$$

Figure 4.26: Type system for full PRE, with the optimization component

of the optimization. We only have to show that the use of safety can not affect soundness in a hazardous way.

Proof. Again we only prove the first half of the theorem. The proof is by induction on the structure of the type derivation and we look at the same cases as for simple PRE.

- Case $:=_{\mathrm{pre}}$: The type derivation is of the form

  $$\overline{\underline{x := a} : spant, spav \longrightarrow spant', spav' \hookrightarrow s_*}$$

  where $spant =_{\mathrm{df}} (spant' \backslash mod(x)) \cup eval(a)) \cap safe$, $spav' =_{\mathrm{df}} ((spav \cup eval(a)) \setminus mod(x)) \cap safe'$.

  We notice that $safe \cup mod(x) \supseteq safe'$ since

  $$\begin{aligned} safe \cup mod(x) &= ant \cup av \cup mod(x) \\ &= (ant' \setminus mod(x)) \cup eval(a) \cup av \cup mod(x) \\ &\supseteq ant' \cup ((av \cup eval(a)) \setminus mod(x)) \\ &= ant' \cup av' \\ &= safe' \end{aligned}$$

From this it follows that $(\mathit{spant} \cap \mathit{spav}) \cup \mathit{eval}(a) \supseteq \mathit{spant}' \cap \mathit{spav}'$:

$$\begin{array}{ll} & (\mathit{spant} \cap \mathit{spav}) \cup \mathit{eval}(a) \\ = & (((\mathit{spant}' \setminus \mathit{mod}(x)) \cup \mathit{eval}(a)) \cap \mathit{spav} \cap \mathit{safe}) \cup \mathit{eval}(a) \\ = & (\mathit{spant}' \setminus \mathit{mod}(x) \cap \mathit{spav} \cap \mathit{safe}) \cup \mathit{eval}(a) \\ \supseteq & \mathit{spant}' \cap (\mathit{spav} \cup \mathit{eval}(a)) \setminus \mathit{mod}(x) \cap \mathit{safe} \\ = & \mathit{spant}' \cap (\mathit{spav} \cup \mathit{eval}(a)) \setminus \mathit{mod}(x) \cap (\mathit{safe} \cup \mathit{mod}(x)) \\ \supseteq & \mathit{spant}' \cap (\mathit{spav} \cup \mathit{eval}(a)) \setminus \mathit{mod}(x) \cap \mathit{safe}' \\ = & \mathit{spant}' \cap \mathit{spav}' \end{array}$$

We also note it separately that $\mathit{spant}' \cap \mathit{spav}' \cap \mathit{mod}(x) = \emptyset$.

The given semantic judgement must be of the form

$$\frac{}{\sigma \succ x := a \to \sigma[x \mapsto [\![a]\!]\sigma]}$$

- Subcase $:=_{1\mathrm{pre}}$: We know that $a \notin \mathit{spav}$. We also know that either $a \notin \mathit{spant}'$ or $x \in FV(a)$ (so $a \notin \mathit{spav}'$), so $a \notin \mathit{spant}' \cap \mathit{spav}'$, which implies $\mathit{spant} \cap \mathit{spav} \supseteq \mathit{spant}' \cap \mathit{spav}'$. Moreover, $s_* =_{\mathrm{df}} x := a$.
  We have the semantic derivation

  $$\frac{}{\sigma_* \succ x := a \to \sigma'_*}$$

  where $\sigma'_* =_{\mathrm{df}} \sigma_*[x \mapsto [\![a]\!]\sigma_*]$. From $\sigma \sim_{\mathit{spant} \cap \mathit{spav}} \sigma_*$ it follows that $[\![a]\!]\sigma = [\![a]\!]\sigma_*$, so that, using $\mathit{spant} \cap \mathit{spav} \supseteq \mathit{spant}' \cap \mathit{spav}'$ as well, we can conclude $\sigma' = \sigma[x \mapsto [\![a]\!]\sigma] \sim_{\mathit{spant}' \cap \mathit{spav}'} \sigma_*[x \mapsto [\![a]\!]\sigma] = \sigma_*[x \mapsto [\![a]\!]\sigma_*] = \sigma'_*$.
- Subcase $:=_{2\mathrm{pre}}$: We have that $a \notin \mathit{spav}$ and a is nontrivial. Also, $s_* =_{\mathrm{df}} \mathit{nv}(a) := a; x := \mathit{nv}(a)$. We have the semantic derivation

  $$\frac{\frac{}{\sigma_* \succ \mathit{nv}(a) := a \to \sigma''_*} \quad \frac{}{\sigma''_* \succ x := \mathit{nv}(a) \to \sigma'_*}}{\sigma_* \succ \mathit{nv}(a) := a; x := \mathit{nv}(a) \to \sigma'_*.}$$

  where $\sigma''_* =_{\mathrm{df}} \sigma_*[\mathit{nv}(a) \mapsto [\![a]\!]\sigma_*]$ and $\sigma'_* =_{\mathrm{df}} \sigma''_*[x \mapsto \sigma''_*(\mathit{nv}(a))] = \sigma''_*[x \mapsto [\![a]\!]\sigma_*]$. From $\sigma \sim_{\mathit{spant} \cap \mathit{spav}} \sigma_*$ it is immediate that $\sigma \sim_{(\mathit{spant} \cap \mathit{spav}) \cup \{a\}} \sigma_*[\mathit{nv}(a) \mapsto [\![a]\!]\sigma] = \sigma_*[\mathit{nv}(a) \mapsto [\![a]\!]\sigma_*] = \sigma''_*$ and therefore by $(\mathit{spant} \cap \mathit{spav}) \cup \{a\} \supseteq \mathit{spant}' \cap \mathit{spav}'$ we have $\sigma' = \sigma[x \mapsto [\![a]\!]\sigma] \sim_{\mathit{spant}' \cap \mathit{spav}'} \sigma''_*[x \mapsto [\![a]\!]\sigma] = \sigma''_*[x \mapsto [\![a]\!]\sigma_*] = \sigma'_*$.
- Subcase $:=_{3\mathrm{pre}}$: We have that $a \in \mathit{spav}$, but then a is nontrivial and $a \in \mathit{safe}$ (as $\mathit{spav} \subseteq \mathit{safe}$), so further $a \in ((\mathit{spant}' \setminus \mathit{mod}(x)) \cup \mathit{eval}(a)) \cap \mathit{safe} = \mathit{spant}$ as well, i.e., $a \in \mathit{spant} \cap \mathit{spav}$. As a consequence, $\mathit{spant} \cap \mathit{spav} \supseteq \mathit{spant}' \cap \mathit{spav}'$, too. We have $s_* =_{\mathrm{df}} x := \mathit{nv}(a)$. We have the semantic derivation

  $$\frac{}{\sigma_* \succ x := \mathit{nv}(a) \to \sigma'_*}$$

where $\sigma'_* =_{\rm df} \sigma_*[x \mapsto \sigma_*(nv(a))]$. From $a \in \mathit{spant} \cap \mathit{spav}$ and $\sigma \sim_{\mathit{spant} \cap \mathit{spav}} \sigma_*$ we learn that $[\![a]\!]\sigma = \sigma_*(nv(a))$. Further, using also that $\mathit{spant} \cap \mathit{spav} \supseteq \mathit{spant}' \cap \mathit{spav}'$, we realize that $\sigma' = \sigma[x \mapsto [\![a]\!]\sigma] \sim_{\mathit{spant}' \cap \mathit{spav}'} \sigma_*[x \mapsto [\![a]\!]\sigma] = \sigma_*[x \mapsto \sigma_*(nv(a))] = \sigma'_*$.

- Case $\text{conseq}_{\text{pre}}$: The type derivation is of the form

$$\dfrac{\begin{array}{c}\vdots\\ \underline{s} : \mathit{spant}_0, \mathit{spav}_0 \longrightarrow \mathit{spant}'_0, \mathit{spav}'_0 \hookrightarrow s_*\end{array}}{\underline{s} : \mathit{spant}, \mathit{spav} \longrightarrow \mathit{spant}', \mathit{spav}' \hookrightarrow s'; s_*; s''}$$

where $(\mathit{spant}, \mathit{spav}) \leq (\mathit{spant}_0, \mathit{spav}_0)$, $(\mathit{spant}'_0, \mathit{spav}'_0) \leq (\mathit{spant}', \mathit{cpav}')$, $s' =_{\rm df} [nv(a) := a \mid a \in (\mathit{spant}_0 \cap \mathit{spav}_0) \setminus \mathit{spav}]$ and $s'' =_{\rm df} [nv(a) := a \mid a \in (\mathit{spant}' \cap \mathit{spav}') \setminus \mathit{spav}'_0]$. First we find a state σ_0 such that $\sigma_* \succ s' \rightarrow \sigma_0$ and $\sigma \sim_{\mathit{spant}_0 \cap \mathit{spav}_0} \sigma_0$. We have the semantic derivation

$$\dfrac{}{\sigma_* \succ s' \rightarrow \sigma_0}$$

where $\sigma_0 =_{\rm df} \sigma_*[nv(a) \mapsto [\![a]\!]\sigma_* \mid a \in (\mathit{spant}_0 \cap \mathit{spav}_0) \setminus \mathit{spav}]$.

As $\sigma \sim_{\mathit{spant} \cap \mathit{spav}} \sigma_*$, it is enough to show that expressions not in $\mathit{spant} \cap \mathit{spav}$ but in $\mathit{spant}_0 \cap \mathit{spav}_0$ are made equal to their corresponding auxiliary variables by s'. But this is exactly what s' does, as

$$\begin{array}{ll} & (\mathit{spant}_0 \cap \mathit{spav}_0) \setminus (\mathit{spant} \cap \mathit{spav}) \\ = & (\mathit{spant}_0 \cap \mathit{spav}_0 \setminus \mathit{spant}) \cup (\mathit{spant}_0 \cap \mathit{spav}_0 \setminus \mathit{spav}) \\ = & (\mathit{spant}_0 \cap \mathit{spav}_0) \setminus \mathit{spav} \end{array}$$

using $\mathit{spant} \supseteq \mathit{spant}_0$.

From the induction hypothesis we obtain that there is a state σ_1 such that $\sigma_0 \succ s_* \rightarrow \sigma_1$ and $\sigma' \sim_{\mathit{spant}'_0 \cap \mathit{spav}'_0} \sigma_1$. It is now enough to show that there is a state σ'_* such that $\sigma_1 \succ s'' \rightarrow \sigma'_*$ and $\sigma' \sim_{\mathit{spav}' \cap \mathit{spav}'} \sigma'_*$. This can be done similarly to the case of s'.

□

For proof transformation, let $P|_{\mathit{spant} \cap \mathit{spav}}$ abbreviate $\bigwedge[\, nv(a) = a \mid a \in \mathit{spant} \cap \mathit{spav}] \wedge P$. We have the following theorem.

Theorem 24 (Preservation of Hoare logic provability/proofs for PRE) *If* $\underline{s} : \mathit{spant}, \mathit{spav} \longrightarrow \mathit{spant}', \mathit{spav}' \hookrightarrow s_*$, *then* $\vdash \{P\}\, s\, \{Q\}$ *implies* $\{P|_{\mathit{spant} \cap \mathit{spav}}\}\, s_*\, \{Q|_{\mathit{spant}' \cap \mathit{spav}'}\}$.

Proof. The proof is by induction on the type derivation and we look at the same cases as before.

- Case $:=_{\text{pre}}$: The type derivation is

$$\frac{}{\underline{x := a} : spant' \cap safe, spav \longrightarrow spant, spav' \cap safe' \hookrightarrow s_*}$$

where $spant' =_{\text{df}} (spant \backslash mod(x) \cup eval(a))$, $spav' =_{\text{df}} (spav \cup eval(a) \backslash mod(x))$. From the soundness proof we remember that $(spant \cap spav) \cup eval(a) \supseteq spant' \cap spav'$ and $spant' \cap spav' \cap mod(x) = \emptyset$. The latter fact tells us that for any a' and P, we have $P[a'/x]|_{spant' \cap spav'} \Leftrightarrow P|_{spant' \cap spav'}[a'/x]$.

The given Hoare logic proof must be of the form

$$\frac{}{\{P[a/x]\}\, x := a\, \{P\}}$$

  - Subcase $:=_{1\text{pre}}$: We have that $a \notin spav$. We also have that either $a \notin spant'$ or $x \in FV(a)$ (so $a \notin spav'$), so altogether $a \notin spant' \cap spav'$, which implies $spant \cap spav \supseteq spant' \cap spav'$. Moreover, $s_* =_{\text{df}} x := a$.

    From $spant \cap spav \supseteq spant' \cap spav'$, it follows that$P[a/x]|_{spant \cap spav} \models P[a/x]|_{spant' \cap spav'}$. The transformed Hoare logic proof is

$$\frac{\dfrac{\dfrac{}{\{P|_{spant' \cap spav'}[a/x]\}\, x := a\, \{P|_{spant' \cap spav'}\}}}{\{P[a/x]|_{spant' \cap spav'}\}\, x := a\, \{P|_{spant' \cap spav'}\}}}{\{P[a/x]|_{spant \cap spav}\}\, x := a\, \{P|_{spant' \cap spav'}\}}$$

  - Subcase $:=_{2\text{pre}}$: We have that $a \notin spav$. We also have that a is nontrivial. Moreover, $s_* =_{\text{df}} nv(a) := a; x := nv(a)$.

    From $(spant \cap spav) \cup \{a\} \supseteq spant' \cap spav'$, it follows that $P[nv(a)/x]|_{(spant \cap spav) \cup \{a\}} \models P[nv(a)/x]|_{spant' \cap spav'}$. From reflexivity of equality, $P[a/x]|_{spant \cap spav} \Leftrightarrow P[a/x]|_{spant \cap spav} \wedge a = a \Leftrightarrow (P[nv(a)/x]|_{(spant \cap spav) \cup \{a\}})[a/nv(a)]$. The transformed Hoare logic proof is

$$\frac{B_0 \quad B_1}{\{P[a/x]|_{spant \cap spav}\}\, nv(a) = a; x := nv(a)\, \{P|_{spant' \cap spav'}\}}$$

    where $B_0 \equiv$

$$\frac{\dfrac{}{\{P[nv(a)/x]|_{(spant \cap spav) \cup \{a\}}[a/nv(a)]\}\, nv(a) = a\, \{P[nv(a)/x]|_{(spant \cap spav) \cup \{a\}}\}}}{\{P[a/x]|_{spant \cap spav}\}\, nv(a) = a\, \{P[nv(a)/x]|_{spant' \cap spav'}\}}$$

    and $B_1 \equiv$

$$\frac{\dfrac{}{\{P|_{spant' \cap spav'}[nv(a)/x]\}\, x := nv(a)\, \{P|_{spant' \cap spav'}\}}}{\{P[nv(a)/x]|_{spant' \cap spav'}\}\, x := nv(a)\, \{P|_{spant' \cap spav'}\}}$$

  - Subcase $:=_{3\text{pre}}$: We have that $a \in spav$. We also have that $a \in spant$(as $a \in spav \subseteq safe$ and a is nontrivial). This has $a \in spant \cap spav$ as

a consequence and therefore we also get $spant \cap spav \supseteq spant' \cap spav'$. Further, $s_* =_{\mathrm{df}} x := nv(a)$.

From $a \in spant \cap spav$ and $spant \cap spav \supseteq spant' \cap spav'$ we get $P[a/x]|_{spant \cap spav} \models P[a/x]|_{spant' \cap spav'} \wedge nv(a) = a$. Substitution of equals for equals gives $P|_{spant' \cap spav'}[a/x] \wedge nv(a) = a \models P|_{spant' \cap spav'}[nv(a)/x]$. The transformed Hoare logic proof is

$$\dfrac{\dfrac{\dfrac{}{\{P|_{spant' \cap spav'}[nv(a)/x]\}\, x := nv(a)\, \{P|_{spant' \cap spav'}\}}}{\{P|_{spant' \cap spav'}[a/x] \wedge nv(a) = a\}\, x := nv(a)\, \{P|_{spant' \cap spav'}\}}}{\dfrac{\{P[a/x]|_{spant' \cap spav'} \wedge nv(a) = a\}\, x := nv(a)\, \{P|_{spant' \cap spav'}\}}{\{P[a/x]|_{spant \cap spav}\}\, x := nv(a)\, \{P|_{spant' \cap spav'}\}}}$$

- Case $\mathrm{conseq}_{\mathrm{pre}}$: The type derivation is

$$\dfrac{\begin{array}{c}\vdots \\ s : spant_0, spav_0 \longrightarrow spant'_0, spav'_0 \hookrightarrow s_*\end{array}}{s : spant, spav \longrightarrow spant', spav' \hookrightarrow s'; s_*; s''}$$

where $(spant, spav) \leq (spant_0, spav_0)$, $(spant'_0, spav'_0) \leq (spant, spav)$ and $s' =_{\mathrm{df}} [nv(a) := a \mid a \in (spant_0 \cap spav_0) \setminus spav]$, $s'' =_{\mathrm{df}} [nv(a) := a \mid a \in (spant' \cap spav') \setminus spav'_0]$. The given Hoare logic proof is

$$\dfrac{\begin{array}{c}\vdots \\ \{P_0\}\, s\, \{Q_0\}\end{array}}{\{P\}\, s\, \{Q\}}$$

where $P \models P_0$ and $Q_0 \models Q$.

By the induction hypothesis, there is a Hoare logic proof of $\{P_0|_{spant_0 \cap spav_0}\}\, s_*\, \{Q_0|_{spant'_0 \cap spav'_0}\}$.

It is an assumption that $P \models P_0$, hence $P|_{spant \cap spav} \models P_0|_{spant \cap spav}$. Using reflexivity of equality we get $P_0|_{spant \cap spav} \Leftrightarrow P_0|_{spant \cap spav} \wedge \bigwedge[a = a \mid a \in (spant_0 \cap spav_0) \setminus (spant \cap spav)] \models P_0|_{spant_0 \cap spav_0}[a/nv(a) \mid a \in (spav_0 \cap spant_0) \setminus (spant \cap spav)]$.

From the soundness proof we remember that $(spant_0 \cap spav_0) \setminus (spant \cap spav) = spant_0 \cap spav_0) \setminus spav$. Hence from the axiom $\{P_0|_{(spant_0 \cap spav_0)}[a/nv(a) \mid a \in (spant_0 \cap spav_0) \setminus spav\,]\}\, s'\, \{P_0|_{(spant_0 \cap spav_0)}\}$ by the consequence rule we have a proof of $\{P|_{(spant \cap spav)}\}\, s'\, \{P_0|_{spant_0 \cap spav_0}\}$.

Similarly we can make a proof of $\{Q_0|_{spant'_0 \cap spav'_0}\}\, s''\, \{Q|_{spant' \cap spav'}\}$.

Putting everything together with the sequence rule, we obtain a proof of $\{P|_{spant \cap spav}\}\, s'; s_*; s''\, \{Q|_{spant' \cap spav'}\}$, which is the required transformed Hoare logic proof.

□

The similarity relation between the states for showing the improvement property is the following. We define $(\sigma, r) \approx_{spant \cap spav, av} (\sigma_*, r_*)$ to mean that two states (σ, r) and (σ_*, r_*) are similar wrt. $spant \cap spav$ and av in the sense that $\sigma \sim_{spant \cap spav} \sigma_*$ and, moreover, $\forall a \in spant \cap spav \setminus av.\, r_*(a) \leq r(a) + 1$ and $\forall a \notin spant \cap spav \setminus av.\, r_*(a) \leq r(a)$.

This relation is more involved than in the case of simple PRE. For simple PRE, an expression being in a type *cpav* meant a "promise" that there will be a use of the expression, because *cpav* implied anticipability. For full PRE this is not the case, since $spant \cap spav$ does not imply that there is a future use of the expression. This is where the notion of *safety* comes into play. Since $spant \cap spav \subseteq safe$ where $safe = ant \cup av$, then if $a \in spant \cap spav$, either $a \in ant$ or $a \in av$. Notice that $a \in spant \cap spav \setminus av$ implies $a \in ant$ which means that we allow an increase in the number of expression evaluations only at places where the expression is anticipable. The reason why there cannot be an increase of evaluations of a where $a \in av$ is simply that availability implies that the expression is already computed on all paths to that program point, so no extra evaluation could have been added. This is the reason why we need to parameterize the similarity relation with av. A simple example explaining this is the program $x := a;$ if b then $y := a$ else skip, which can be optimized to $nv(a) = a; x := nv(a);$ if b then $y := nv(a)$ else skip. After the first statement, a is clearly both safely partially available and safely partially anticipable. But the count of evaluations of a should not be one larger for the optimized program than for the original one, hence we need to take into account that a is available. Consequently we can guarantee that the final number of evaluations of a for the optimized program is no bigger than that of the original one.

Theorem 25 (Improvement property of full PRE)
If $\underline{s} : spant, spav \longrightarrow spant', spav' \hookrightarrow s_*$ *and* $(\sigma, r) \approx_{(spant \cap spav, av)} (\sigma_*, r_*)$*, then*
— $(\sigma, r) \succ s \rightarrow (\sigma', r')$ *implies the existence of* (σ'_*, r'_*) *such that* $(\sigma', r') \approx_{(spant' \cap spav', av')} (\sigma'_*, r'_*)$ *and* $(\sigma_*, r_*) \succ s_* \rightarrow (\sigma'_*, r'_*)$*,*
— $(\sigma_*, r_*) \succ s_* \rightarrow (\sigma'_*, r'_*)$ *implies the existence of* (σ', r') *such that* $(\sigma', r') \approx_{(spant' \cap spav', av')} (\sigma'_*, r'_*)$ *and* $(\sigma, r) \succ s \rightarrow (\sigma', r')$*.*

Proof. We only prove the first half of theorem. The proof is by induction on the structure of the type derivation. We look at the following nontrivial cases.

- Case $:=_{\text{pre}}$: The type derivation is of the form

  $$\frac{}{\underline{x := a} : spant, spav \longrightarrow spant', spav' \hookrightarrow s_*}$$

  where $spant =_{\text{df}} ((spant' \setminus mod(x)) \cup eval(a)) \cap safe$, $spav' =_{\text{df}} (spav \cup eval(a)) \setminus mod(x) \cap safe'$.

We have that $\mathit{spant} \cap \mathit{spav} \setminus \mathit{av} \subseteq (\mathit{spant}' \cap \mathit{spav}' \setminus \mathit{av}') \cup \mathit{eval}(a)$:

$$\begin{array}{ll}
\multicolumn{2}{l}{(\mathit{spant}' \cap \mathit{spav}' \setminus \mathit{av}') \cup \mathit{eval}(a)} \\
= & (\mathit{spant}' \cap (\mathit{spav} \cup \mathit{eval}(a)) \setminus \mathit{mod}(x) \cap \mathit{safe}' \setminus ((\mathit{av} \cup \mathit{eval}(a)) \setminus \mathit{mod}(x))) \\
& \cup \mathit{eval}(a) \\
= & (\mathit{spant}' \cap (\mathit{spav} \cup \mathit{eval}(a)) \setminus \mathit{mod}(x) \cap \mathit{safe}' \setminus (\mathit{av} \cup \mathit{eval}(a))) \cup \mathit{eval}(a) \\
= & (\mathit{spant}' \cap \mathit{spav} \setminus \mathit{mod}(x) \cap \mathit{safe}' \setminus \mathit{av}) \cup \mathit{eval}(a) \\
\supseteq & (\mathit{spant}' \cap \mathit{spav} \setminus \mathit{mod}(x) \cap \mathit{safe} \setminus \mathit{av}) \cup \mathit{eval}(a) \\
\supseteq & ((\mathit{spant}' \setminus \mathit{mod}(x)) \cup \mathit{eval}(a)) \cap \mathit{safe} \cap \mathit{spav} \setminus \mathit{av} \\
= & \mathit{spant} \cap \mathit{spav} \setminus \mathit{av}
\end{array}$$

From the proof of soundness we remember that $(\mathit{spant} \cap \mathit{spav}) \cup \mathit{eval}(a) \supseteq \mathit{spant}' \cap \mathit{spav}'$. We also recall that $\mathit{spant}' \cap \mathit{spav}' \cap \mathit{mod}(x) = \emptyset$. This yields that that $(\mathit{spant} \cap \mathit{spav} \setminus \mathit{av}) \cup \mathit{eval}(a) \supseteq ((\mathit{spant} \cap \mathit{spav}) \cup \mathit{eval}(a)) \setminus \mathit{av} \supseteq (\mathit{spant}' \cap \mathit{spav}') \setminus \mathit{av} \supseteq \mathit{spant}' \cap \mathit{spav}' \setminus ((\mathit{av} \cup \mathit{eval}(a)) \setminus \mathit{mod}(x)) = \mathit{spant}' \cap \mathit{spav}' \setminus \mathit{av}'$.

So for any nontrivial expression $a' \neq a$, $a' \in \mathit{spant} \cap \mathit{spav} \setminus \mathit{av}$ and $a' \in \mathit{spant}' \cap \mathit{spav}' \setminus \mathit{av}'$ are equivalent.

The given semantic derivation must be of the form

$$\frac{}{(\sigma, r) \succ x := a \rightarrow (\sigma[x \mapsto [\![a]\!]\sigma], r[a \mapsto r(a) + 1])}$$

- Subcase $:=_{1\mathrm{pre}}$: We have that $a \notin \mathit{spav}$, which implies that $a \notin \mathit{spant} \cap \mathit{spav} \setminus \mathit{av}$, and either $a \notin \mathit{spant}'$ or $x \in FV(a)$ (so $a \notin \mathit{spav}'$), which implies that $a \notin \mathit{spant}' \cap \mathit{spav}' \setminus \mathit{av}'$. Moreover, $s_* =_{\mathrm{df}} x := a$.

  We have the semantic derivation

  $$\frac{}{(\sigma_*, r_*) \succ x := a \rightarrow (\sigma'_*, r'_*)} \ .$$

  where $(\sigma'_*, r'_*) =_{\mathrm{df}} (\sigma_*[x \mapsto [\![a]\!]\sigma_*], r_*[a \mapsto r_*(a) + 1])$. From the assumption we know that $r_*(a) \leq r(a)$, so $r'_*(a) = r_*(a) + 1 \leq r(a) + 1 = r'(a)$.

- Subcase $:=_{2\mathrm{pre}}$: We have that $a \notin \mathit{spav}$, implying $a \notin \mathit{spant} \cap \mathit{spav} \setminus \mathit{av}$, and both $a \in \mathit{spant}'$ and $x \notin FV(a)$, hence $a \in \mathit{av}'$, implying $a \notin \mathit{spant}' \cap \mathit{spav}' \setminus \mathit{av}'$. Moreover, $s_* =_{\mathrm{df}} \mathit{nv}(a) := a; x := \mathit{nv}(a)$. We have the semantic derivation

  $$\frac{\dfrac{}{(\sigma_*, r_*) \succ \mathit{nv}(a) := a \rightarrow \sigma''_*, r''_*} \qquad \dfrac{}{(\sigma''_*, r''_*) \succ x := \mathit{nv}(a) \rightarrow (\sigma'_*, r'_*)}}{(\sigma_*, r_*) \succ \mathit{nv}(a) := a; x := \mathit{nv}(a) \rightarrow (\sigma'_*, r'_*)} \ .$$

  where $(\sigma''_*, r''_*) =_{\mathrm{df}} (\sigma_*[\mathit{nv}(a) \mapsto [\![a]\!]\sigma_*], r_*[a \mapsto r_*(a) + 1])$ and $(\sigma'_*, r'_*) =_{\mathrm{df}} (\sigma''_*[x \mapsto \sigma''_*(\mathit{nv}(a))], r''_*)$. Similarly to the previous case, from $r_*(a) \leq r(a)$ we obtain $r'_*(a) = r''_*(a) = r_*(a) + 1 \leq r(a) + 1 = r'(a)$.

– Subcase $:=_{3\mathrm{pre}}$: We have that $a \in \mathit{spav}$, which has as a consequence that $a \in \mathit{spant}$ (as a is nontrivial and $a \in \mathit{spav} \subseteq \mathit{safe}$), but we do not know whether $a \notin \mathit{av}$. Hence we do not know whether $a \in \mathit{spant} \cap \mathit{spav} \setminus \mathit{av}$. However, since $a \in \mathit{spav}'$ if and only if $a \in \mathit{av}'$ (both hold if $x \notin FV(a)$ and neither holds if $x \in FV(a)$), we know that $a \notin \mathit{spant}' \cap \mathit{spav}' \setminus \mathit{av}'$. Moreover, $s_* =_{\mathrm{df}} x := \mathit{nv}(a)$. We have the semantic derivation

$$\frac{}{(\sigma_*, r_*) \succ x := \mathit{nv}(a) \rightarrow (\sigma'_*, r'_*)} \ .$$

where $(\sigma'_*, r'_*) =_{\mathrm{df}} (\sigma_*[x \mapsto \sigma_*(\mathit{nv}(a))], r_*)$. From $r_*(a) \leq r(a) + 1$ (which holds both if $a \notin \mathit{spant} \cap \mathit{spav} \setminus \mathit{av}$ and if $a \in \mathit{spant} \cap \mathit{spav} \setminus \mathit{av}$) we get that $r'_*(a) = r_*(a) \leq r(a) + 1 = r'(a)$.

In all three subcases, for any nontrivial $a' \neq a$, if $a' \in \mathit{spant} \cap \mathit{spav} \setminus \mathit{av}$, then we have that $a' \in \mathit{spant}' \cap \mathit{spav}' \setminus \mathit{av}'$ as well and therefore from $r_*(a') \leq r(a') + 1$ we get $r'_*(a') = r_*(a') \leq r(a') + 1 = r'(a') + 1$. Similarly, for $a' \neq a$ such that $a' \notin \mathit{spant} \cap \mathit{spav} \setminus \mathit{av}$ we have $a' \notin \mathit{spant}' \cap \mathit{spav}' \setminus \mathit{av}'$, so $r_*(a') \leq r(a')$ gives us $r'_*(a') = r_*(a') \leq r(a') = r'(a')$.

- Case $\mathrm{conseq}_{\mathrm{pre}}$: The type derivation is of the form

$$\frac{\begin{array}{c}\vdots\\ \underline{s} : \mathit{spant}_0, \mathit{spav}_0 \longrightarrow \mathit{spant}'_0, \mathit{spav}'_0 \hookrightarrow s_*\end{array}}{\underline{s} : \mathit{spant}, \mathit{spav} \longrightarrow \mathit{spant}', \mathit{spav}' \hookrightarrow s'; s_*; s''}$$

where $(\mathit{ant}, \mathit{av}) \leq (\mathit{ant}_0, \mathit{av}_0)$, $(\mathit{spant}, \mathit{spav}) \leq (\mathit{spant}_0, \mathit{spav}_0)$, $(\mathit{ant}'_0, \mathit{av}'_0) \leq (\mathit{ant}', \mathit{av}')$, $(\mathit{spant}'_0, \mathit{spav}'_0) \leq (\mathit{spant}', \mathit{spav}')$ and $s' =_{\mathrm{df}} [\mathit{nv}(a) := a \mid a \in (\mathit{spant}_0 \cap \mathit{spav}_0) \setminus \mathit{spav}]$, $s'' =_{\mathrm{df}} [\mathit{nv}(a) := a \mid a \in (\mathit{spant}' \cap \mathit{spav}') \setminus \mathit{spav}'_0]$. First we find a state (σ_0, r_0) such that $(\sigma_*, r_*) \succ s' \rightarrow (\sigma_0, r_0)$ and $(\sigma, r) \approx_{\mathit{spant}_0 \cap \mathit{spav}_0, \mathit{av}_0} (\sigma_0, r_0)$. We have the semantic derivation

$$\frac{}{(\sigma_*, r_*) \succ s' \rightarrow (\sigma_0, r_0)} \ .$$

where $(\sigma_0, r_0) =_{\mathrm{df}} (\sigma_*[\mathit{nv}(a) \mapsto [\![a]\!]\sigma_* \mid a \in (\mathit{spant}_0 \cap \mathit{spav}_0) \setminus \mathit{spav}], r_*[a \mapsto r_*(a) + 1 \mid a \in (\mathit{spant}_0 \cap \mathit{spav}_0) \setminus \mathit{spav}])$ and we know that $(\sigma, r) \approx_{\mathit{spant} \cap \mathit{spav}, \mathit{av}} (\sigma_*, r_*)$.

For any expression $a \in \mathit{spav} \setminus \mathit{av}$, from $\mathit{spav} \setminus \mathit{av} \subseteq \mathit{spav} \setminus \mathit{av}_0 \subseteq \mathit{safe} \setminus \mathit{av}_0 \subseteq \mathit{ant}_0 \subseteq \mathit{spant}_0$, $\mathit{spav} \subseteq \mathit{spav}_0$ and $\mathit{av} \supseteq \mathit{av}_0$, we have $a \in \mathit{spant}_0 \cap \mathit{spav}_0 \setminus \mathit{av}_0$, whereas from $r_*(a) \leq r(a) + 1$ (which is necessarily guaranteed) we can conclude $r_0(a) = r_*(a) \leq r(a) + 1$. For any expression $a \in \mathit{spav} \cap \mathit{av}$, we have $a \notin \mathit{spant} \cap \mathit{spav} \setminus \mathit{av}$, so from $r_*(a) \leq r(a)$ we can conclude $r_0(a) = r_*(a) \leq r(a)$ (which is sufficient to ensure).

For any nontrivial expression $a \notin \mathit{spant}_0 \cap \mathit{spav}_0$, it is obvious that $a \notin \mathit{spant}_0 \cap \mathit{spav}_0 \setminus \mathit{av}_0$ and from $\mathit{spav} \setminus \mathit{av} \subseteq \mathit{spav} \setminus \mathit{av}_0 \subseteq \mathit{safe} \setminus \mathit{av}_0 \subseteq \mathit{ant}_0 \subseteq \mathit{spant}_0$ and

$spav \subseteq spav_0$ we learn that $a \notin spant \cap spav \setminus av$. In this situation $r_*(a) \leq r(a)$ tells us that $r_0(a) = r_*(a) \leq r(a)$.

If an expression a is in $(spant_0 \cap spav_0) \setminus spav$, then $a \notin spant \cap spav \setminus av$ and $a \in spant_0 \cap spav_0 \setminus av_0$ (as $a \notin spav \supseteq av \supseteq av_0$), thus from $r_*(a) \leq r(a)$ we can conclude $r_0(a) = r_*(a) + 1 \leq r(a) + 1$.

By the induction hypothesis, there must exist a state (σ_1, r_1) such that $(\sigma_0, r_0) \succ s_* \rightarrow (\sigma_1, r_1)$ and $(\sigma', r') \approx_{spant'_0 \cap spav'_0, av'_0} (\sigma_1, r_1)$. It is now enough to exhibit a state (σ'_*, r'_*) such that $(\sigma_1, r_1) \succ s'' \rightarrow (\sigma'_*, r'_*)$ and $(\sigma', r') \approx_{spant' \cap spav', av'} (\sigma'_*, r'_*)$. This can be done in the same way as for s'.

$\square$

4.5 Related work

The work in this chapter is closely related to the work by Laud et al. [38], which gives a general method for casting monotone forward and backward data-flow analyses as type systems. We followed their scheme of translating dataflow analyses into type systems, but concentrated our work on soundness and proof transformations for specific optimizations, rather than the meta-theoretic properties of the translation scheme. Also, differently from their work, we look at optimizations which require multiple analyses. It is related in the same vein to the work of Nielson and Nielson on flow logic [49], which is an approach to static analyses to separate the specification of when an analysis acceptable for a program from computing that analysis information.

One of the first papers considering proof transformation in the context of compilation was by Barthe et al. [11]. They showed that in the absence of program optimizations, proof transformation can be simply identity (modulo variable renaming and other minor changes), by constructing a weakest precondition calculus for the low-level language which generates proof obligations that are syntactically similar to those generated from the original, high-level program. In a continuation of this work, Barthe et al. [8] considered an optimizing compiler and showed how certificates for high-level programs can be translated into certificates for the compiled programs. In their approach, only assertion strengthening is considered. As a result, optimizations which are sound for similarity relations weaker than equality on the original program variables (such as dead code elimination) need to be addressed in an ad hoc fashion. For dead code elimination, dead assignments are not removed, but the register transfer language they use is extended with "ghost assignments" that do not affect the standard semantics of the RTL (being equivalent to skip), but are treated by the verification condition generator as if they were normal assignments. In the most recent work in this direction [9], Barthe and Kunz develop an abstract interpretation framework for certificate translation, where they also address the issue of assertion weakening.

Also close to our work is that of Benton [13], which describes constant folding and dead code elimination as a type system, but does not consider optimizations such as common subexpression elimination or partial redundancy elimination. He also employs the relational method for stating the soundness of analyses and optimizations. Additionally, he develops a relational Hoare logic for proving instances of soundness of a program optimization, by showing that a given relation holds between the original and the optimized program. Metatheoretic statements about optimizations still need to be proved as statements about the logic.

Lerner et al. [39, 40] have developed a framework for writing compiler optimizations that can be automatically proved sound. They separate the proof that an analysis is sound into two parts: the analysis-dependent part where they define a soundness property which must be satisfied by each propagation rule, but can be typically proved automatically and the analysis-independent part, which can be proved manually once and for all.

There is a large body of work dedicated on proving optimizations correct. Bertot et al. report a correctness proof of compiler optimizations where optimizations are formulated as instances of a general framework for data flow analysis, implemented in the Coq proof assistant [16]. Lacey et al. describe a framework in which optimizations are specified as rewrite rules with side conditions that are written as temporal logic formulas [37]. Proof of the correctness of the optimization can then be made using the temporal logic formula as an assumption, which significantly simplifies the proof.

The work on translation validation has a goal similar to ours, the main idea being that rather than formally verifying an optimizing compiler in full, one would validate the correctness of the target program wrt. source program after every program compiler run [46, 70]. This is similar to our goal, where we translate the program proof for each particular program, rather than having a general argument about the soundness of the optimization. Their main motivation for instance-based validation is tackling the complexity of full compiler verification, while in the context of PCC, the goal is to prevent having to ship the source code to the consumer as evidence that target program is correct.

Another related line of work is by Albert et al. [1] on abstraction carrying code, where analysis results are certified in an abstract interpretation framework rather than via type systems.

In Section 4.4 we showed that PRE is actually improving and how resource usage proofs can be transformed just like functional correctness proofs. This is related to work by Aspinall et al. [5] on optimization validation, which, orthogonally to translation validation, aims to show that an instance of an optimizing transformation actually improves the program wrt. some resource usage measure (as opposed to preserving the semantics).

Partial redundancy elimination is a very subtle and sophisticated analysis, as can be seen by the vast amount of literature dedicated to getting it right. The optimization was first proposed by Morel and Renvoise [44], using a bidirectional

analysis. Biderectional analyses were also used by subsequent papers by Dhamdhere et al., addressing the shortcomings of the original algorithm [24, 22]. A new interest in PRE was sparked by papers by Knoop et al. [33, 26], which showed that bidirectional analyses are not needed for PRE, but the optimization can be performed by cascading four unidirectional analysis. The algorithm, called lazy code motion finds code insertion points by modelling the optimization as a code motion transformation. Many implementations of PRE in compilers use a slight modification of this algorithm by Drechsler and Stadel [25]. Later algorithms avoid the explicit code motion modelling step by identifying code insertion points directly [50, 23, 17]. The most recent development in this field by Xue and Knoop [69] show PRE to be a maximum flow problem. Our work is based on the formulation by Paleri et al. [50], which seems to be the simplest and most straightforward algorithm, but for some reason has not received very much attention in literature.

4.6 Conclusion

The main goal of this chapter was to demonstrate how the type-systematic approach to dataflow analysis can be used for showing soundness of optimizations and as a result, also for performing proof transformations which follow from the soundness proofs.

On the example of dead code elimination, we showed that by using our approach, it is completely straightforward to transform proofs where soundness of the optimization depends on similarity relation weaker than equality on program variables. Common subexpression elimination was technically interesting for the fact that it requires links (essentially use-def chains) to be established across the phrase structure of a program, which is inherently non-compositional, since it is expressed in terms of a control flow graph. We showed that this can be overcome with a combined type systems. The scalability of the type-systematic approach was further demonstrated on the example of partial redundancy elimination, a subtle and complicated optimization that performs code motion and edge splitting to place moved expression evaluations. In our type system, this corresponds to introducing new assignments at subsumption inferences.

In addition, we believe this chapter to be a good advocacy for using type systems for describing dataflow analysis in general. Type systems provide a clean separation of concerns of what is a valid analysis result and how one is computed. In fact, it is unnecessary to know exactly about the algorithm to believe the analysis result. On the other hand, extracting a type inference algorithm from the type-system is in general straightforward. In certain PCC settings, it can also be important to communicate analysis results via checkable certificates. In this case, the type derivation of a program can be compressed into annotations from which reconstruction of the full derivation is simple, the important piece of information being the loop invariants. The type-system presentation of an optimization also makes it easy to show that the optimization is generally sound, via a simple structural induction on type deriva-

tions, which can make our approach beneficial even when communicating analysis results or transforming proofs is not required.

On the example of PRE, we looked briefly into the optimality argument and showed that PRE improves a program in a nonstrict sense, i.e., it does not make it worse. In general, it is the best improvement one can achieve, since an already optimal program cannot be improved further. It would be interesting, however, to explore special cases where an optimization leads to a strict improvement. In this case, it would be be possible to also show the improvement in the resource usage proof, i.e., that the program uses less resources after the optimization.

It would also be interesting to look at how to state that the optimization is the best possible one. This would require a strict definition of what can be considered a valid modification of a program: for example, one would not expect an optimization to change the algorithmic behavior of a program, even if the program's semantics is preserved. Then an optimal modification would be a sound acceptable modification improving at least as much as any other.

One road that was left unexplored here concerns modularity of analyses and optimizations. For common subexpression and partial redundancy elimination, we proved the optimization to be sound in one monolithic step. Instead, this could have been done in independent and reusable steps. For CSE, it could be done in three steps: by first proving that available expressions type system is sound, then proving conditional partial anticipability sound with respect to any sound available expressions analysis, and by finally defining and proving common subexpression elimination sound with respect to any sound conditional partial anticipability analysis. For showing soundness of PRE, the soundness proof of availability could then be reused. On the other hand, it is also easy to argue for the benefit of the monolithic proof. The monolithic proof deals directly with standard big-step semantics, while the modular proof would require non-standard, instrumented semantics to be introduced. This would be an extra set of formalisms to have to trust, which somewhat defeats the purpose of the modular proof (which is to provide a cleaner, more understandable proof). But it would certainly be of interest to look into designing a systematic framework for cascaded semantic arguments and proof transformations about such cascaded optimizations.

Chapter 5

Bytecode transformations

So far, we have looked at transformations that operate on high-level structured code. Dataflow analyses however are typically performed on CFG-based program descriptions. Also, so far we have only looked at transformations which require unidirectional analyses, i.e., analyses which propagate information in only one direction in the control flow path. Although many analyses are of this form, there are many useful bidirectional ones (as already mentioned, the original PRE algorithms were all bidirectional, as are several type inference algorithms).

In this chapter, we investigate some bytecode transformations which address exactly these two issues. The unstructured nature of bytecode directs us to the use of CFG based description of the analyses (and thus the proof transformation) and the presence of stacks requires many non-trivial analyses to be bidirectional.

We also spell out several algorithms for direct optimization of stack-based code, which to our knowledge have not appeared in the literature so far.

5.1 Background

Popular Java compilers such as Sun's `javac` or Eclipse's Java Compiler are very conservative with optimizations, since most optimizations are presumed to be performed in the virtual machine by the just-in-time compiler. On the other hand, ahead-of-time optimizations are still important, especially in the context of mobile devices, where just-in-time compilers are not as powerful as on desktops or servers [21] and the size of the distributed binaries should be as small as possible.

Optimizing bytecode directly offers challenges not present in high or intermediate-level program optimizations. The following reasons can be outlined:

- Expressions and statements are not explicit. In a naive approach, a reconstruction of expression trees from instructions would be required for many optimizations.
- Related instructions are not necessarily next to each other. A value could be put on the stack, a number of other instructions executed and only then the value used and popped. This means that related instructions, for example

those that put a value on a stack, and those that consume it, can be arbitrarily far apart. Links between them need to be found during the analysis.

- A single expression can span several different basic blocks. The Java Virtual Machine specification does not require zero stack depth at control flow junctions, so an expression used in a basic block can be partially computed in other basic blocks.

Most work done in the area of intraprocedural optimizations takes the approach of only optimizing code inside basic blocks [35, 41, 67, 59]. There are probably two main reasons for this. First, analyzing bytecode across basic block boundaries is significantly more subtle than analyzing only code inside basic blocks. Second, in compiled code, expressions that span basic blocks are rare (although they do arise, e.g., from Java's ?-expressions). Some prominent bytecode optimizers, such as Soot [64], use an approach where class files are first converted to three-address intermediate representation, optimized using standard techniques and converted back to bytecode. A similar approach is employed in many JIT compilers.

In this chapter we give uniform formal declarative descriptions and algorithms for a number of optimizations and their underlying analyses for the stack-based language presented in Chapter 2. We conflate the semantic domain $\mathbb{Z} + \mathbb{B}$ of values into a single domain of representations of integers and booleans ("words") on which both the arithmetic and boolean operations are total, so operand type errors cannot occur. Stack underflow errors are still possible. Alternatively, we could assume that programs come well-typed. We will show how the transformations can be proven sound and that they preserve program proofs. Just like for the structured language, we use type systems with a transformation component for this purpose.

The analyses and optimizations we address are dead store, load-pop pairs, duplicating loads and store-load pairs elimination, which are typical optimization situations in stack-based code. The optimizations are designed to work on general code, i.e., they do not make any assumptions about its form (the code need not be the compiled version of a high-level program). Also, the analyses and optimizations are not in any way "intra-basic block". On the contrary, they work across basic block boundaries and do not require that the stack is empty at these. We show that optimizations modifying pairs of instructions crossing basic block boundaries require bidirectional analyses, as information must be propagated both forward and backward during an analysis.

5.2 Dead code elimination

As was explained in Chapter 4, standard dead code elimination optimization removes those statements from a program which do not affect the values of variables that are live at the end of the program. The optimization is easy to perform on high-level programs or intermediate (expressions-based low-level) code after a live variables analysis by removing assignments to variables which are known to be dead

immediately after the assignment. In stack-based code, where expressions are not explicit and related instructions are not necessarily next to each other, removing dead code is not so straightforward. For example the program $x := z + y$ could be compiled into

$0,$ load z
$1,$ load y
$2,$ add
$3,$ store x
$4,$

If the analysis shows that x is dead, then in the intermediate code, the assignment to x can be deleted. In the stack-based code however, not only the store instruction on line 3, but also lines 0-2 should be deleted.

Another issue which sets stack-based code optimizations apart from optimizations in the intermediate language is that statements and expression can span several basic blocks or, put in another way, basic blocks are not necessarily entered into or exited from with an empty stack. A simple example of such code is the following stack-based low-level equivalent of if b then $x := z$ else $y := z$ where z is loaded only once, and in both branches only the store instruction is applied:

$0,$ load z
$1,$ load b
$2,$ gotoF 5
$3,$ store x
$4,$ goto 6
$5,$ store y
$6,$

If live variable analysis reveals that the variable x is dead, the store instruction at line 3 cannot simply be removed, since if the true branch were taken, the unassigned value of the variable x would be left on the stack after exiting the branch. Also, the load instruction on line 0 cannot be deleted, because, while it is used by a dead store, it is also used by a live store in the false branch. In such cases, without moving instructions, the best thing to do is replacing instruction 3 with a pop.

We approach dead code elimination in two stages. In the first stage (which we call dead stores elimination), all dead store and add instructions are replaced with pop instructions (so that the optimization does not affect the stack height at any label) based on an analogue of the standard live variables analysis. In the second stage, pop instructions with corresponding preceding load/push instruction(s) are eliminated, if possible, and care is taken that stack heights remain consistent after this transformation.

Both of these analyses are completely general, do not make any assumptions on the form of bytecode, and work across basic block boundaries.

5.2.1 Dead stores elimination

The live variables analysis of our stack-based language is similar to live variables analysis for languages with expressions, except that the stack is also accounted for. This means that in addition to variables being possibly live or certainly dead, stack positions can also be either possibly live or certainly dead. For example, if we know that immediately after an execution of a $\mathsf{store}\ x$ instruction, the variable x is dead, then we also know that before the execution the top of the stack is dead, meaning that whatever value the stack top holds, it does not affect the value of a live variable.

We describe both the analysis and the optimization in terms of a type system.

For live variable analysis, the code type $\Sigma \in \mathbf{CodeType}$ is an assignment of a label type to every label: $\mathbf{CodeType} =_{\mathrm{df}} \mathbf{Label} \to \mathbf{LabelType}$. A label type is a pair of a stack and store type: $\mathbf{LabelType} =_{\mathrm{df}} \mathbf{StackType} \times \mathbf{StoreType}$. Stack types $es \in \mathbf{StackType}$ are defined by the grammar

$$es \quad ::= \quad [] \mid e :: es \mid *$$

where $e \in \mathbf{LocType}$ is a location type "possibly live" or "certainly dead": $\mathbf{LocType} =_{\mathrm{df}} \{L, D\}$. The stack type $*$ stands for stacks of arbitrary height with all stack positions dead. Store types $d \in \mathbf{StoreType}$ are assignments of variables to location types: $\mathbf{StoreType} =_{\mathrm{df}} \mathbf{Var} \to \mathbf{LocType}$.

The subtyping and typing rules are given in Figure 5.1. We use the shorthand Σ_ℓ for $\Sigma(\ell)$. The subtyping judgement $\Sigma_\ell \leq \Sigma'_\ell$ denotes that the first label type is a subtype of the second (i.e., stronger). Similarly, the subtyping judgement $\Sigma \leq \Sigma'$ denotes that the first code type is a subtype of the other. The typing judgement $\Sigma \vdash (\ell, \mathit{instr})$ signifies that the labelled instruction (ℓ, instr) admits type Σ, i.e., that Σ is a valid analysis of this particular labelled instruction (independent of any possible code context in which it may occur). The typing judgement $\Sigma \vdash c$ means that the code c types with Σ, i.e., that Σ is a valid analysis of c as a whole.

The typing rules of our particular analysis only allow a variable or stack position to be marked "dead" at a label ℓ in a valid code type if there cannot be a path from ℓ to a label ℓ' such that the instruction at ℓ' contains a useful use of that position and the variable or stack position is not redefined on the path. Otherwise it must be marked "live". A typical useful use of the stack top is storing it in a variable marked "live" at the successor label. The stack top and next-to-top positions are usefully used by an addition, provided the stack top is marked "live" at the successor label. The stack top used by a pop is certainly dead, since its value is lost and does not affect the values of any location. For $\mathsf{load}\ x$, the type of x depends on its type and the type of the stack top at the successor label: if x was live at the successor label already, it stays live, otherwise if the top of the stack was live (thus needed), x also becomes live.

The type system also has a transformation component for transforming a given piece of code into an optimized variant, guided by a valid code type. The judgement $\Sigma \vdash c \hookrightarrow c'$ denotes that, based on a valid analysis Σ, c can be optimized to c'.

$$\frac{}{L \leq D} \qquad \frac{}{e \leq e} \qquad \frac{}{[] \leq []} \qquad \frac{}{es \leq *} \qquad \frac{e \leq e' \quad es \leq es'}{e :: es \leq e' :: es'} \qquad \frac{\forall x. d(x) \leq d'(x)}{d \leq d'}$$

$$\frac{es \leq es' \quad d \leq d'}{(es, d) \leq (es', d')} \qquad \frac{\forall \ell \in \mathbf{Label}. \Sigma_\ell \leq \Sigma'_\ell}{\Sigma \leq \Sigma'}$$

$$\frac{\Sigma_\ell \leq (L : es, d[x \mapsto D]) \quad (es, d) = \Sigma_{\ell+1} \quad d(x) = L}{\Sigma \vdash (\ell, \mathsf{store}\ x) \hookrightarrow (\ell, \mathsf{store}\ x)}\ \text{store}_1$$

$$\frac{\Sigma_\ell \leq (D : es, d) \quad (es, d) = \Sigma_{\ell+1} \quad d(x) = D}{\Sigma \vdash (\ell, \mathsf{store}\ x) \hookrightarrow (\ell, \mathsf{pop})}\ \text{store}_2$$

$$\frac{\Sigma_\ell \leq (es, d[x \mapsto d(x) \wedge e]) \quad (e : es, d) \leq \Sigma_{\ell+1}}{\Sigma \vdash (\ell, \mathsf{load}\ x) \hookrightarrow (\ell, \mathsf{load}\ x)}\ \text{load} \qquad \frac{\Sigma_\ell \leq (es, d) \quad (e : es, d) \leq \Sigma_{\ell+1}}{\Sigma \vdash (\ell, \mathsf{push}\ n) \hookrightarrow (\ell, \mathsf{push}\ n)}\ \text{push}$$

$$\frac{\Sigma_\ell \leq (L :: L :: es, d) \quad (L :: es, d) = \Sigma_{\ell+1}}{\Sigma \vdash (\ell, \mathsf{add}) \hookrightarrow (\ell, \mathsf{add})}\ \text{add}_1 \qquad \frac{\Sigma_\ell \leq (D :: D :: es, d) \quad (D :: es, d) \leq \Sigma_{\ell+1}}{\Sigma \vdash (\ell, \mathsf{add}) \hookrightarrow (\ell, \mathsf{pop})}\ \text{add}_2$$

$$\frac{\Sigma_\ell \leq (D : es, d) \quad (es, d) = \Sigma_{\ell+1}}{\Sigma \vdash (\ell, \mathsf{pop}) \hookrightarrow (\ell, \mathsf{pop})}\ \text{pop} \qquad \frac{\Sigma_\ell \leq ((e_0 \wedge e_1) :: es, d) \quad (e_0 :: e_1 :: es, d) \leq \Sigma_{\ell+1}}{\Sigma \vdash (\ell, \mathsf{dup}) \hookrightarrow (\ell, \mathsf{dup})}\ \text{dup}$$

$$\frac{\Sigma_\ell \leq \Sigma_m}{\Sigma \vdash (\ell, \mathsf{goto}\ m) \hookrightarrow (\ell, \mathsf{goto}\ m)}\ \text{goto} \qquad \frac{\Sigma_\ell \leq (L :: es, d) \quad (es, d) = \Sigma_{\ell+1} \wedge \Sigma_m}{\Sigma \vdash (\ell, \mathsf{gotoF}\ m) \hookrightarrow (\ell, \mathsf{gotoF}\ m)}\ \text{gotoF}$$

$$\frac{\forall \ell \in \text{dom}(c).\, \Sigma \vdash (\ell, c_\ell) \hookrightarrow (\ell, c'_\ell)}{\Sigma \vdash c \hookrightarrow c'}\ \text{code}$$

Figure 5.1: Type system for live variables analysis and dead stores elimination

The instructions that can be optimized (those that decrease the stack height by 1) have two rules. A $\mathsf{store}\ x$ instruction can be optimized, if x is marked "dead" in the posttype (i.e., at the successor label of the instruction). An add instruction can be optimized, if the top of the stack is "dead" in the posttype.

Note that while we have not spelled them out, optimizations for load and push instructions could also be added to the type system. Namely, if in the posttype of a load instruction the top of the stack is "dead", it is obvious that the value is not used on any forward path, thus the concrete value put on the stack does not matter (only the correct stack height does). Thus, the instruction could be replaced with the cheapest instruction that puts some value of the correct type on the stack, e.g., $\mathsf{push}\ 0$.

To illustrate that the analysis does not need related instructions to be next to each other, let us look at the program $y := x; x := w + 1$ as a piece of code where the assignment $y := x$ has been moved to the middle (instructions 3-4). For this example, we assume the variable y to be live and variable x to be dead at the end of the program. The analysis gives the following result.

Σ_ℓ		ℓ, c_ℓ	ℓ, c'_ℓ
$[]$	$[y \mapsto D, x \mapsto L]$	$0, \mathsf{push}\ 1$	$0, \mathsf{push}\ 1$
$[D]$	$[y \mapsto D, x \mapsto L]$	$1, \mathsf{load}\ w$	$1, \mathsf{load}\ w$
$[D, D]$	$[y \mapsto D, x \mapsto L]$	$2, \mathsf{add}$	$2, \mathsf{pop}$
$[D]$	$[y \mapsto D, x \mapsto L]$	$3, \mathsf{load}\ x$	$3, \mathsf{load}\ x$
$[L, D]$	$[y \mapsto D, x \mapsto D]$	$4, \mathsf{store}\ y$	$4, \mathsf{store}\ y$
$[D]$	$[y \mapsto L, x \mapsto D]$	$5, \mathsf{store}\ x$	$5, \mathsf{pop}$
$[]$	$[y \mapsto L, x \mapsto D]$	$6,$	$6,$

The optimization replaces the store x and add instructions with pop instructions (since in both cases variable x resp. the stack top is dead in the type of the successor). This leaves the stack balanced. Our next analysis (Subsection 5.2.2) will show that the pop instructions on lines 2 and 5 can be removed together with the instructions on lines 0 and 1, since stack usage will remain consistent after those transformations too.

Principal type inference algorithm for dead stores We now look at the principal type inference algorithm which finds the weakest valid code type that is stronger than a given one, using the given one as the initial value (Algorithm 1). Typically, the given code type would set the stack type to be empty and the store to be "all live" at the exit labels (successor labels outside the domain). Elsewhere, the label type has the default value "any".

input : Bytecode program c, intial type Σ^{init}
output: Final type Σ
$\Sigma \leftarrow \Sigma^{\text{init}}$;
repeat
  $\Sigma^{\text{old}} \leftarrow \Sigma$;
  foreach $\ell \in \text{dom}(c)$ **do**
    updbwd(c_ℓ);
  end
until $\Sigma \leq \Sigma^{\text{old}}$;

Algorithm 1: Dead stores

The (partial) greatest lower bound operations are as follows:

glbs of value types			glb of store types		
$\top$	$=_{\text{df}}$	D	$\top(x)$	$=_{\text{df}}$	D
$D \wedge D$	$=_{\text{df}}$	D	$(d \wedge d')(x)$	$=_{\text{df}}$	$d(x) \wedge d'(x)$
$L \wedge e$	$=_{\text{df}}$	L			
$e \wedge L$	$=_{\text{df}}$	L			

glb of stack types		
$\top$	$=_{\text{df}}$	$*$
$* \wedge es$	$=_{\text{df}}$	es
$es \wedge *$	$=_{\text{df}}$	es
$[] \wedge []$	$=_{\text{df}}$	$[]$
$[] \wedge e :: es$	$=_{\text{df}}$	error
$e :: es \wedge []$	$=_{\text{df}}$	error
$e :: es \wedge e' :: es'$	$=_{\text{df}}$	$(e \wedge e') :: (es \wedge es')$

glbs of label types		
$\top$	$=_{\text{df}}$	$(\top, \top)$
$(es, d) \wedge (es', d')$	$=_{\text{df}}$	$(es \wedge es', d \wedge d')$

The update procedure is as follows:

c_ℓ	$\text{updbwd}_\ell, \Sigma_\ell :=$
store x	case $\Sigma^{\text{old}}_{\ell+1}$ of $(es, d) \mapsto \Sigma_\ell \wedge (d(x) :: es, d[x \mapsto D])$
load x	case $\Sigma^{\text{old}}_{\ell+1}$ of $(*, d) \mapsto \Sigma_\ell$ $([], d) \mapsto$ error $(e :: es, d) \mapsto \Sigma_\ell \wedge (es, d[x \mapsto d(x) \wedge e])$
push n	case $\Sigma^{\text{old}}_{\ell+1}$ of $(*, d) \mapsto \Sigma_\ell$ $([], d) \mapsto$ error $(e :: es, d) \mapsto \Sigma_\ell \wedge (es, d)$
add	case $\Sigma^{\text{old}}_{\ell+1}$ of $(*, d) \mapsto \Sigma_\ell \wedge (D :: D :: *, d)$ $([], d) \mapsto$ error $(e :: es, d) \mapsto \Sigma_\ell \wedge (e :: e :: es, d)$
pop	case $\Sigma^{\text{old}}_{\ell+1}$ of $(es, d) \mapsto \Sigma_\ell \wedge (D :: es, d)$
dup	case $\Sigma^{\text{old}}_{\ell+1}$ of $(*, d) \mapsto \Sigma_\ell \wedge (D :: *, d)$ $([], d) \mapsto$ error $([e], d) \mapsto$ error $(e_0 :: e_1 :: es, d) \mapsto \Sigma_\ell \wedge ((e_0 \wedge e_1) :: es, d)$
goto m	$\Sigma_\ell \wedge \Sigma^{\text{old}}_m$
gotoF m	case $\Sigma^{\text{old}}_{\ell+1} \wedge \Sigma^{\text{old}}_m$ of $(es, d) \mapsto \Sigma_\ell \wedge (L :: es, d)$

The optimization is easily stated to be sound using a label-type indexed similarity relation on states, $\sim_{es}$. The relations are defined as follows:

$$\frac{}{[] \sim_{[]} []} \qquad \frac{zs \sim_{es} zs_*}{z :: zs \sim_{L::es} z :: zs_*} \qquad \frac{zs \sim_{es} zs_*}{z :: zs \sim_{D::es} z_* :: zs_*} \qquad \frac{}{zs \sim_* zs_*}$$

$$\frac{\forall x \in \mathbf{Var}. d(x) = L \Rightarrow \sigma(x) = \sigma_*(x)}{\sigma \sim_d \sigma_*} \qquad \frac{zs \sim_{es} zs_* \quad \sigma \sim_d \sigma_*}{(\ell, zs, \sigma) \sim_{(es,d)} (\ell, zs_*, \sigma_*)}$$

We can see that two states are related by a label type (a stack and store type), if they agree on the locations marked "live". Reducing an original piece of code and

its optimized form from a related pair of states must maintain this relation. We obtain the following soundness theorem.

Theorem 26 (Soundness of dead stores elimination) *If $\Sigma \vdash c \hookrightarrow c'$, then:*

1. *If $(\ell, zs, \sigma) \sim_{\Sigma_\ell} (\ell_*, zs_*, \sigma_*)$, then*

   — for any (ℓ', zs', σ') such that $c \vdash (\ell, zs, \sigma) \twoheadrightarrow (\ell', zs', \sigma')$ there exist $(\ell'_*, zs'_*, \sigma'_*)$ such that $(\ell', zs', \sigma') \sim_{\Sigma_{\ell'}} (\ell'_*, zs'_*, \sigma'_*)$ and $c' \vdash (\ell_*, zs_*, \sigma_*) \twoheadrightarrow (\ell'_*, zs'_*, \sigma'_*)$,

   — for any $(\ell'_*, zs'_*, \sigma'_*)$ such that $c' \vdash (\ell_*, zs_*, \sigma_*) \twoheadrightarrow (\ell'_*, zs'_*, \sigma'_*)$ there exist (ℓ', zs', σ') such that $(\ell', zs', \sigma') \sim_{\Sigma_{\ell'}} (\ell'_*, zs'_*, \sigma'_*)$ and $c \vdash (\ell, zs, \sigma) \twoheadrightarrow (\ell', zs', \sigma')$.

2. *If $(\ell, zs, \sigma) \sim_{\Sigma_\ell} (\ell_*, zs_*, \sigma_*)$, then*

   — for any (ℓ', zs', σ') such that $c \vdash (\ell, zs, \sigma) \twoheadrightarrow^* (\ell', zs', \sigma') \not\twoheadrightarrow$ there exist $(\ell'_*, zs'_*, \sigma'_*)$ such that $(\ell', zs', \sigma') \sim_{\Sigma_{\ell'}} (\ell'_*, zs'_*, \sigma'_*)$ and $c' \vdash (\ell_*, zs_*, \sigma_*) \twoheadrightarrow^* (\ell'_*, zs'_*, \sigma'_*) \not\twoheadrightarrow$.

   — for any $(\ell'_*, zs'_*, \sigma'_*)$ such that $c' \vdash (\ell_*, zs_*, \sigma_*) \twoheadrightarrow^* (\ell'_*, zs'_*, \sigma'_*) \not\twoheadrightarrow$ there exist (ℓ', zs', σ') such that $(\ell', zs', \sigma') \sim_{\Sigma_{\ell'}} (\ell'_*, zs'_*, \sigma'_*)$ and $c \vdash (\ell, zs, \sigma) \twoheadrightarrow^* (\ell', zs', \sigma') \not\twoheadrightarrow$.

Proof. Part (1) is easily checked by inspecting the typing/transformation rules for labelled instructions, the two interesting cases are the optimizations of $\mathsf{store}\ x$ and add. We look at the first half and assume that $(\ell, zs, \sigma) \sim_{\Sigma_\ell} (\ell_*, zs_*, \sigma_*)$ and $c \vdash (\ell, zs, \sigma) \twoheadrightarrow (\ell', zs', \sigma')$.

- Case

  $$\frac{\Sigma_\ell \leq (D : es, d) \quad (es, d) = \Sigma_{\ell+1} \quad d(x) = D}{\Sigma \vdash (\ell, \mathsf{store}\ x) \hookrightarrow (\ell, \mathsf{pop})}\ \mathrm{store}_2$$

  We know that $\Sigma_{\ell+1} = (es, d)$ and $\Sigma_\ell \leq (D : es, d)$. From Lemma 1, the only possible reduction is $c \vdash (\ell, n :: zs, \sigma) \twoheadrightarrow (\ell + 1, zs, \sigma[x \mapsto n])$. The corresponding reduction for c'_ℓ is $c' \vdash (\ell, n :: zs_*, \sigma_*) \twoheadrightarrow (\ell + 1, zs_*, \sigma_*)$. Since x is dead in $\Sigma_{\ell+1}$, we have that since $\sigma \sim_{\Sigma_\ell} \sigma_*$ then also $\sigma[x \mapsto n] \sim_{\Sigma_{\ell+1}} \sigma_*$. Also, if $n :: zs \sim_{\Sigma_\ell} n :: zs_*$ then $zs \sim_{\Sigma_{\ell+1}} zs_*$. Thus it is easy to see that $(\ell + 1, zs, \sigma[x \mapsto n]) \sim_{\Sigma_{\ell+1}} (\ell + 1, zs_*, \sigma_*)$.

- Case

  $$\frac{\Sigma_\ell \leq (D :: D :: es, d) \quad (D :: es, d) \leq \Sigma_{\ell+1}}{\Sigma \vdash (\ell, \mathsf{add}) \hookrightarrow (\ell, \mathsf{pop})}\ \mathrm{add}_2$$

  We follow a similar reasoning as in the above case. We know that $(D :: es, d) = \Sigma_{\ell+1}$ and $\Sigma_\ell \leq (D :: D :: es, d)$. From Lemma 1, the only possible reduction is $c \vdash (\ell, n :: m :: zs, \sigma) \twoheadrightarrow (\ell + 1, n + m :: zs, \sigma)$. The corresponding reduction for c'_ℓ is $c' \vdash (\ell, n :: m :: zs_*, \sigma_*) \twoheadrightarrow (\ell + 1, m :: zs_*, \sigma_*)$. If $(\ell, n :: m :: zs, \sigma) \sim_{\Sigma_\ell} (\ell, n :: m :: zs_*, \sigma_*)$, then $(\ell + 1, n + m :: zs, \sigma) \sim_{\Sigma_{\ell+1}} (\ell + 1, m :: zs_*, \sigma_*)$, since states σ and σ_* are not changed in the reduction and the top of the stack is dead in $\Sigma_{\ell+1}$, so we only have to observe that $n + m$ has the same type as m.

The rest of the cases are trivial.

Part (2) follows from part (1) by induction on the length of the reduction sequence. □

Proof transformation for dead store elimination We now look at proof optimizations of PUSH programs. Since the type system is non-compositional, we use the non-compositional logic introduced in Chapter 2. While it would be possible to apply proof transformation for the compositional logic described in Chapter 3, the non-compositional logic is simpler to use for presentation purposes.

The general idea here is similar to that of Hoare logic proof transformation. Each assertion at a label is modified wrt. the type at that label, which reflects the changes made in the program. For dead store elimination, a statement about dead variables or stack positions might become unprovable, since the code affecting them could be replaced. Thus, we should quantify out all program variables and stack positions which are dead.

Let $P|_\Sigma$ be P where all P_ℓ have been modified wrt. Σ_ℓ. Let $P_\ell|_{\Sigma_\ell} =_{\mathrm{df}} \exists[v(x) \mid d(x) = D]\exists[v(st(n)) \mid es(n) = D]P_\ell[v(x)/x \mid d(x) = D][v(st(n))/st(n) \mid es(n) = D]$ where $(es, d) = \Sigma_\ell$ and v is some assignment of unique logic variable names to program variables and stack positions. Intuitively, it means existentially quantifying out variables and stack positions which are dead. As an example, for assertion $st(0) = 5 \wedge x = 4 \wedge y = 5$ and type $([D], [x \mapsto L, y \mapsto D])$, the new assertion would be $\exists s_0, y'.s_0 = 5 \wedge x = 4 \wedge y' = 5$, which is equivalent to $x = 5$.

We will use the following lemma in the proof of program proof preservation.

Lemma 11 $\mathsf{shift}(P)|_{(D:es,d)} \equiv \mathsf{shift}(P|_{(es,d)})$

We have the following theorem for proof preservation.

Theorem 27

1. *If* $\Sigma \vdash (\ell, instr) \hookrightarrow (\ell, instr')$ *and* $P \vdash (\ell, instr)$ *then* $P|_\Sigma \vdash (\ell, instr')$
2. *If* $\Sigma \vdash c \hookrightarrow c'$ *and* $P \vdash c$*, then also* $P|_\Sigma \vdash c'$*.*

Proof. For part (1), we give a constructive proof by examining each typing rule. The interesting cases are store and add.

- Case store$_2$. The typing judgement is of the following form

$$\frac{\Sigma_\ell \leq (D : es, d) \quad (es, d) = \Sigma_{\ell+1} \quad d(x) = D}{\Sigma \vdash (\ell, \mathsf{store}\ x) \hookrightarrow (\ell, \mathsf{pop})}\ \mathrm{store}_2$$

  We also have the logic judgement of the form

$$\frac{P_\ell \models \mathsf{shift}(P_{\ell+1})[st(0)/x]}{P \vdash (\ell, \mathsf{store}\ x)}$$

We have to show that

$$\frac{P_\ell|_{\Sigma_\ell} \models \mathsf{shift}(P_{\ell+1}|_{\Sigma_{\ell+1}})}{P|_\Sigma \vdash (\ell, \mathsf{pop})}$$

We have the following sequence of entailments:

$P_\ell\|_{\Sigma_\ell}$	$\models$	$\mathsf{shift}(P_{\ell+1})[st(0)/x]\|_{\Sigma_\ell}$	by the given entailment for store and existential introduction and elimination
	$\models$	$\mathsf{shift}(P_{\ell+1})[st(0)/x]\|_{(D::es,d)}$	by $\Sigma_\ell \leq (D :: es, d)$
	$\models$	$\mathsf{shift}(P_{\ell+1})\|_{(D::es,d)}$	both x and $st(0)$ dead, the witness for $st(0)$ serves as witness for $v(x)$
	$\models$	$\mathsf{shift}(P_{\ell+1}\|_{(es,d)})$	Lemma 11
	$\models$	$\mathsf{shift}(P_{\ell+1}\|_{\Sigma_{\ell+1}})$	

- Case add$_2$. The typing judgement is of the following form

$$\frac{\Sigma_\ell \leq (D :: D :: es, d) \quad (D :: es, d) \leq \Sigma_{\ell+1}}{\Sigma \vdash (\ell, \mathsf{add}) \hookrightarrow (\ell, \mathsf{pop})}$$

We also have the logic judgement of the form

$$\frac{P_\ell \models \mathsf{shift}(P_{\ell+1})[st(0) + st(1)/st(1)]}{P \vdash (\ell, \mathsf{add})}$$

We have to show that

$$\frac{P_\ell|_{\Sigma_\ell} \models \mathsf{shift}(P_{\ell+1}|_{\Sigma_{\ell+1}})}{P|_\Sigma \vdash (\ell, \mathsf{pop})}$$

We have the following sequence of entailments

$P_\ell|_{\Sigma_\ell} \models \mathsf{shift}(P_{\ell+1})[st(0) + st(1)/st(1)]_{\Sigma_\ell}$
by the given entailment for add and
by existential introduction and elimination
$\models \mathsf{shift}(P_{\ell+1})[st(0) + st(1)/st(1)]|_{(D::D::es,d)}$
by $\Sigma_\ell \leq (D :: D :: es, d)$
$\models \mathsf{shift}(P_{\ell+1})|_{(D::D::es,d)}$
$st(0)$ and $st(1)$ are dead
$\models \mathsf{shift}(P_{\ell+1}|_{(D::es,d)})$
Lemma 11
$\models \mathsf{shift}(P_{\ell+1}|_{\Sigma_{\ell+1}})$

Part (2) follows trivially from part (1) by the definition of $P \vdash c$.

□

Example

We now look at the example presented in Chapter 4 for dead code elimination for the high-level language. The to-be-optimized program is while $x < y$ do $(x := x * 2; z := z + 1)$. The given post-type for its compiled version states that the stack should be empty, and z and y should be dead at the end of the program. The program is compiled the following way and obtains the type given below. The optimized version is also given.

ℓ		Σ_ℓ	c_ℓ	c'_ℓ
1	$[]$	$[x \mapsto L, y \mapsto L, z \mapsto D]$	load x	load x
2	$[L]$	$[x \mapsto L, y \mapsto L, z \mapsto D]$	load y	load y
3	$[L, L]]$	$[x \mapsto L, y \mapsto L, z \mapsto D]$	less	less
4	$[L]$	$[x \mapsto L, y \mapsto L, z \mapsto D]$	gotoF 14	gotoF 14
5	$[]$	$[x \mapsto L, y \mapsto L, z \mapsto D]$	load x	load x
6	$[L]$	$[x \mapsto L, y \mapsto L, z \mapsto D]$	push 2	push 2
7	$[L, L]$	$[x \mapsto L, y \mapsto L, z \mapsto D]$	mult	mult
8	$[L]$	$[x \mapsto L, y \mapsto L, z \mapsto D]$	store x	store x
9	$[]$	$[x \mapsto L, y \mapsto L, z \mapsto D]$	load z	load z
10	$[D]$	$[x \mapsto L, y \mapsto L, z \mapsto D]$	push 1	push 1
11	$[D, D]$	$[x \mapsto L, y \mapsto L, z \mapsto D]$	add	$\underline{\text{pop}}$
12	$[D]$	$[x \mapsto L, y \mapsto L, z \mapsto D]$	store z	$\underline{\text{pop}}$
13	$[]$	$[x \mapsto L, y \mapsto L, z \mapsto D]$	goto 1	goto 1
14	$[]$	$[x \mapsto L, y \mapsto D, z \mapsto D]$		

The pre- and post-type of the program and assertions associated with each label are given below. The concrete entailments between assertions have not been spelled out directly, but they should be obvious.

Assertion	ℓ	Instruction
$Pre \equiv z = 1 \wedge x = 2 \wedge y > 1$		
$x < y \wedge 2 * x = 2^{z+1} \wedge z + 1 \leq ceil(\log y) \vee \neg(x < y) \wedge ...$	1	load y
$x < st(0) \wedge 2 * x = 2^{z+1} \wedge z + 1 \leq ceil(\log y) \vee \neg(x < st(0)) \wedge ...$	2	load x
$st(0) < st(1) \wedge 2 * x = 2^{z+1} \wedge z + 1 \leq ceil(\log y) \vee \neg(st(0) < st(1)) \wedge ...$	3	less
$st(0) \wedge 2 * x = 2^{z+1} \wedge z + 1 \leq ceil(\log y) \vee \neg(st(0)) \wedge ...$	4	gotoF 14
$2 * x = 2^{z+1} \wedge z + 1 \leq ceil(\log y)$	5	load x
$2 * st(1) = 2^{z+1} \wedge z + 1 \leq ceil(\log y)$	6	push 2
$st(0) * st(1) = 2^{z+1} \wedge z + 1 \leq ceil(\log y)$	7	mult
$st(0) = 2^{z+1} \wedge z + 1 \leq ceil(\log y)$	8	store x
$x = 2^{z+1} \wedge z + 1 \leq ceil(\log y)$	9	push 1
$x = 2^{z+st(0)} \wedge z + st(0) \leq ceil(\log y)$	10	load z
$x = 2^{st(0)+st(1)} \wedge st(0) + st(1) \leq ceil(\log y)$	11	add
$x = 2^{st(0)} \wedge st(0) \leq ceil(\log y)$	12	store z
$x = 2^{z} \wedge z \leq ceil(\log y)$	13	goto 1
$Post \equiv x = 2^{z} \wedge z = ceil(\log y)$	14	

The transformed assertions are the following, where each dead variable and stack position is quantified out.

Assertion		Instruction
$Pre \equiv z = 1 \wedge x = 2 \wedge y > 1$		
$\exists z'.x < y \wedge 2 * x = 2^{z'+1} \wedge z' + 1 \leq ceil(\log y) \vee \neg(x < y) \wedge ...$	1	load y
$\exists z'.(x < st(0) \wedge 2 * x = 2^{z'+1} \wedge z' + 1 \leq ceil(\log y) \vee \neg(x < st(0)) \wedge ...$	2	load x
$\exists z'.st(0) < st(1) \wedge 2 * x = 2^{z'+1} \wedge z' + 1 \leq ceil(\log y) \vee \neg(st(0) < st(1)) \wedge ...$	3	less
$\exists z'.st(0) \wedge 2 * x = 2^{z'+1} \wedge z' + 1 \leq ceil(\log y) \vee \neg(st(0)) \wedge ...$	4	gotoF 14
$\exists z'.2 * x = 2^{z'+1} \wedge z' + 1 \leq ceil(\log y)$	5	load x
$\exists z'.2 * st(1) = 2^{z'+1} \wedge z' + 1 \leq ceil(\log y)$	6	push 2
$\exists z'.st(0) * st(1) = 2^{z'+1} \wedge z' + 1 \leq ceil(\log y)$	7	mult
$\exists z'.st(0) = 2^{z'+1} \wedge z' + 1 \leq ceil(\log y)$	8	store x
$\exists z'.x = 2^{z'+1} \wedge z' + 1 \leq ceil(\log y)$	9	push 1
$\exists z', st0.x = 2^{z'+st0} \wedge z' + st0 \leq ceil(\log y)$	10	load z
$\exists st0, st1.x = 2^{st0+st1} \wedge st0 + st1 \leq ceil(\log y)$	11	pop
$\exists st0.x = 2^{st0} \wedge st0 \leq ceil(\log y)$	12	pop
$\exists z'.x = 2^{z'} \wedge z' \leq ceil(\log y)$	13	goto 1
$Post \equiv x = 2^{z} \wedge z = ceil(\log y)$	14	

5.2.2 Load-pop pairs elimination

This analysis tries to find pop instructions with corresponding load/push instructions and eliminate them. The optimization introduces a subtlety that is present in all bytecode transformations which remove pairs of stack height changing instructions across basic block boundaries. This is illustrated in Figure 5.2 (where the ls nodes denote level sequences of instructions).[1] Looking at this example, it might seem that the load x instruction can be eliminated together with pop. Closer examination reveals that this is not the case: since load y is used by store z, the pop instruction cannot be removed, because then, after taking branch 2, the stack would not be balanced. This in turn means that load x cannot be removed. As can be seen from this example, a unidirectional analysis is not enough to come to such conclusion: information that a stack position is definitely needed flows backward from store z to load y along branch 3, but then the same information flows forward along path 2, and again backward along path 1. Thus a bidirectional analysis is needed, which at each node propagates information both forward and backward. We also see that we are not really dealing with pairs, but webs of instructions in general.

In the appropriate type system, a code type $\Sigma \in$ **CodeType** is again an assignment of a label type to every label: **CodeType** $=_{\mathrm{df}}$ **Label** $\rightarrow$ **LabelType**. Here, label types are stack types, **LabelType** $=_{\mathrm{df}}$ **StackType**. Stack types es are defined by the grammar

$$es \quad ::= \quad [] \mid e :: es \mid *$$

where $e \in$ **LocType** are "mandatory" and "optional" elements: **LocType** $=_{\mathrm{df}}$ $\{\mathrm{mnd}, \mathrm{opt}\}$. The stack type $*$ is for stacks of arbitrary height with all positions optional.

[1] A sequence of instructions is a *level sequence*, if the net change of the stack height by these instructions is 0 and the instructions do not consume any values that were already present in the stack before executing these instructions.

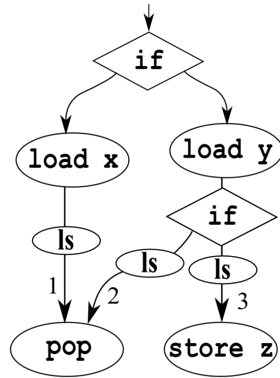


Figure 5.2: Example program

The typing and subtyping rules are given in Figure 5.3. The typing rules state that, if at some label a stack element is marked “mandatory”, then at all other labels of its lifetime, this particular element is also considered “mandatory”. Thus the typing rules explain which optimizations are acceptable. The rule for store instructions states that the instruction always requires a “mandatory” element on the stack, thus its predecessors must definitely leave a value on top of the stack. Instructions that put elements on the stack “do not care”: if an element is required, they can push a value (a mnd element on the stack in the posttype), otherwise the instruction could be omitted (an opt element on the stack in the posttype). The same holds for pop: if an element is definitely left on the stack, a pop instruction is not removed, otherwise it can be removed.

The analysis (i.e., type derivation) algorithm, as mentioned above, is bidirectional. The intuition behind the algorithm is the following. The types definitely required at some labels should be given (typically the types at the exit labels of the code are set to be the empty stack, possibly also the type at the entry label). All other types are initialized to the default type “any state”. The algorithm then computes the weakest valid type of the code that is stronger than the given type. At each label, information is gathered from all its successors and predecessors. The constraints initiate from the given types and the store and conditional jump instructions, which require that a value is present on the stack for them (i.e., a mnd element has to be on top of the stack type). Other instructions that produce or consume a value from the stack can initially be assumed to produce or consume “useless” values (denoted by opt in the type system). Type information arriving from different directions to a program point can be intersected according to subtyping relations given in Figure 5.3. This guarantees that, if an instruction definitely needs a value on the stack, this information is propagated to all of its predecessors. Similarly, if an instruction definitely must produce a value on the stack (since some subsequent instruction may need it), this information is propagated to its successor.

$$\frac{}{\text{mnd} \le \text{opt}} \qquad \frac{}{e \le e} \qquad \frac{}{[] \le []} \qquad \frac{e \le e' \quad es \le es'}{e :: es \le e' :: es'} \qquad \frac{}{es \le *} \qquad \frac{\forall \ell \in \textbf{Label}. \Sigma_\ell \le \Sigma'_\ell}{\Sigma \le \Sigma'}$$

$$\frac{\Sigma_\ell = \text{mnd} :: \Sigma_{\ell+1}}{\Sigma \vdash (\ell, \mathsf{store}\ x) \hookrightarrow (\ell, \mathsf{store}\ x)}\ \text{store}$$

$$\frac{\text{mnd} :: \Sigma_\ell = \Sigma_{\ell+1}}{\Sigma \vdash (\ell, \mathsf{load}\ x) \hookrightarrow (\ell, \mathsf{load}\ x)}\ \text{load}_1 \qquad \frac{\text{opt} :: \Sigma_\ell = \Sigma_{\ell+1}}{\Sigma \vdash (\ell, \mathsf{load}\ x) \hookrightarrow (\ell, \mathsf{nop})}\ \text{load}_2$$

$$\frac{\text{mnd} :: \Sigma_\ell = \Sigma_{\ell+1}}{\Sigma \vdash (\ell, \mathsf{push}\ n) \hookrightarrow (\ell, \mathsf{push}\ n)}\ \text{push}_1 \qquad \frac{\text{opt} :: \Sigma_\ell = \Sigma_{\ell+1}}{\Sigma \vdash (\ell, \mathsf{push}\ n) \hookrightarrow (\ell, \mathsf{nop})}\ \text{push}_2$$

$$\frac{\Sigma_\ell = \text{mnd} :: \text{mnd} :: es \quad \text{mnd} :: es = \Sigma_{\ell+1}}{\Sigma \vdash (\ell, \mathsf{add}) \hookrightarrow (\ell, \mathsf{add})}\ \text{add}_1 \qquad \frac{\Sigma_\ell = \text{opt} :: \text{opt} :: es \quad \text{opt} :: es = \Sigma_{\ell+1}}{\Sigma \vdash (\ell, \mathsf{add}) \hookrightarrow (\ell, \mathsf{nop})}\ \text{add}_2$$

$$\frac{\Sigma_\ell = \text{mnd} :: \Sigma_{\ell+1}}{\Sigma \vdash (\ell, \mathsf{pop}) \hookrightarrow (\ell, \mathsf{pop})}\ \text{pop}_1 \qquad \frac{\Sigma_\ell = \text{opt} :: \Sigma_{\ell+1}}{\Sigma \vdash (\ell, \mathsf{pop}) \hookrightarrow (\ell, \mathsf{nop})}\ \text{pop}_2$$

$$\frac{\Sigma_\ell = \text{mnd} :: es \quad \text{mnd} :: \text{mnd} :: es = \Sigma_{\ell+1}}{\Sigma \vdash (\ell, \mathsf{dup}) \hookrightarrow (\ell, \mathsf{dup})}\ \text{dup}_1 \qquad \frac{\Sigma_\ell = e_1 :: es \quad \text{opt} :: e_1 :: es = \Sigma_{\ell+1}}{\Sigma \vdash (\ell, \mathsf{dup}) \hookrightarrow (\ell, \mathsf{nop})}\ \text{dup}_2$$

$$\frac{\Sigma_\ell = \Sigma_m}{\Sigma \vdash (\ell, \mathsf{goto}\ m) \hookrightarrow (\ell, \mathsf{goto}\ m)}\ \text{goto} \qquad \frac{\Sigma_\ell = \text{mnd} :: es \quad es = \Sigma_m \quad es = \Sigma_{\ell+1}}{\Sigma \vdash (\ell, \mathsf{gotoF}\ m) \hookrightarrow (\ell, \mathsf{gotoF}\ m)}\ \text{gotoF}$$

$$\frac{\forall \ell \in \text{dom}(c).\, \Sigma \vdash (\ell, c_\ell) \hookrightarrow (\ell, c_{*\ell})}{\Sigma \vdash c \hookrightarrow c_*}\ \text{code}$$

Figure 5.3: Type system for load-pop pairs elimination

Looking at the example in Figure 5.2, where we initialize the posttype of the control-flow graph to be the empty stack, the pretype of store z requires a mnd element on the top of the stack. This means that the posttype of load y has to have mnd on the top of the stack. This information propagates to the pop instruction, and from there to load x instruction. Thus the analysis shows that no instruction can be deleted. If, on the other hand, store z were not present, the postlabels of the two load instructions and prelabel of the pop instruction would keep their initial opt stack top types, and the three instructions could be deleted.

A piece of code corresponding to Figure 5.2 is given in the following example in the left column (where the level sequences of instructions have been omitted to simplify presentation). It gets a type showing that no optimization is possible. In the right column, we consider a minimally different piece of code where store z instruction has been replaced with pop. Here the analysis shows that both the pop and corresponding load instructions can be removed.

Σ_ℓ	ℓ, c_ℓ	ℓ, c'_ℓ
$[]$	$0,$ load b	$0,$ load b
$[\mathrm{mnd}]$	$1,$ gotoF 9	$1,$ gotoF 9
$[]$	$2,$ load y	$2,$ load y
$[\mathrm{mnd}]$	$3,$ load b'	$3,$ load b'
$[\mathrm{mnd}, \mathrm{mnd}]$	$4,$ gotoF 7	$4,$ gotoF 7
$[\mathrm{mnd}]$	$5,$ store z	$5,$ store z
$[]$	$6,$ goto 11	$6,$ goto 11
$[\mathrm{mnd}]$	$7,$ pop	$7,$ pop
$[]$	$8,$ goto 11	$8,$ goto 11
$[]$	$9,$ load x	$9,$ load x
$[\mathrm{mnd}]$	$10,$ goto 7	$10,$ goto 7
$[]$	$11,$	$11,$

Σ_ℓ	ℓ, c_ℓ	ℓ, c'_ℓ
$[]$	$0,$ load b	$0,$ load b
$[\mathrm{mnd}]$	$1,$ gotoF 9	$1,$ gotoF 9
$[]$	$2,$ load y	$2,$ nop
$[\mathrm{opt}]$	$3,$ load b'	$3,$ load b'
$[\mathrm{mnd}, \mathrm{opt}]$	$4,$ gotoF 7	$4,$ gotoF 7
$[\mathrm{opt}]$	$5,$ pop	$5,$ nop
$[]$	$6,$ goto 11	$6,$ goto 11
$[\mathrm{opt}]$	$7,$ pop	$7,$ nop
$[]$	$8,$ goto 11	$8,$ goto 11
$[]$	$9,$ load x	$9,$ nop
$[\mathrm{opt}]$	$10,$ goto 7	$10,$ goto 7
$[]$	$11,$	$11,$

Principal type inference Algorithm 2 calculates the greatest type of a piece of code c smaller than a given code type Σ_{init} (with $\Sigma^{\mathrm{init}}_\ell = \top$ for most ℓ), if such exists, iterating forward and backward update procedures:

input : Bytecode program c, intial type Σ^{init}
output: Final type Σ
$\Sigma \leftarrow \Sigma^{\mathrm{init}}$;
repeat
  $\Sigma^{\mathrm{old}} \leftarrow \Sigma$;
  foreach $\ell \in \mathrm{dom}(c)$ **do**
    $\mathrm{updfwd}(c_\ell)$;
    $\mathrm{updbwd}(c_\ell)$;
  end
until $\Sigma^{\mathrm{old}} \leq \Sigma$;

Algorithm 2: Load-pop pairs

The (partial) glb operations are as follows:

glbs of value types		
$\top$	$=_{\mathrm{df}}$	opt
$\mathrm{opt} \wedge \mathrm{opt}$	$=_{\mathrm{df}}$	opt
$\mathrm{mnd} \wedge e$	$=_{\mathrm{df}}$	mnd
$e \wedge \mathrm{mnd}$	$=_{\mathrm{df}}$	mnd

glbs of label types		
$\top$	$=_{\mathrm{df}}$	$*$
$* \wedge es$	$=_{\mathrm{df}}$	es
$es \wedge *$	$=_{\mathrm{df}}$	es
$[] \wedge []$	$=_{\mathrm{df}}$	$[]$
$[] \wedge (e :: es)$	$=_{\mathrm{df}}$	error
$(e :: es) \wedge []$	$=_{\mathrm{df}}$	error
$(e :: es) \wedge (e' :: es')$	$=_{\mathrm{df}}$	$(e \wedge e') :: (es \wedge es')$

The update procedures are as follows:

c_ℓ	$\mathrm{updfwd}_\ell, \Sigma_{\ell+1} :=$	$\mathrm{updbwd}_\ell, \Sigma_\ell :=$
store x	case $\Sigma_\ell^{\mathrm{old}}$ of $* \mapsto \Sigma_{\ell+1}$ $[] \mapsto$ error $e :: es \mapsto \Sigma_{\ell+1} \wedge es$	case $\Sigma_{\ell+1}^{\mathrm{old}}$ of $es \mapsto \Sigma_\ell \wedge \mathrm{mnd} :: es$
load x	case $\Sigma_\ell^{\mathrm{old}}$ of $es \mapsto \Sigma_{\ell+1} \wedge \mathrm{opt} :: es$	case $\Sigma_{\ell+1}^{\mathrm{old}}$ of $* \mapsto \Sigma_\ell$ $[] \mapsto$ error $e :: es \mapsto \Sigma_\ell \wedge es$
push n	case $\Sigma_\ell^{\mathrm{old}}$ of $es \mapsto \Sigma_{\ell+1} \wedge \mathrm{opt} :: es$	case $\Sigma_{\ell+1}^{\mathrm{old}}$ of $* \mapsto \Sigma_\ell$ $[] \mapsto$ error $e :: es \mapsto \Sigma_\ell \wedge es$
add	case $\Sigma_\ell^{\mathrm{old}}$ of $* \mapsto \Sigma_{\ell+1} \wedge \mathrm{opt} :: *$ $[] \mapsto$ error $[e] \mapsto$ error $e_0 :: e_1 :: es \mapsto$ $\Sigma_{\ell+1} \wedge (e_0 \wedge e_1) :: es$	case $\Sigma_{\ell+1}^{\mathrm{old}}$ of $* \mapsto \Sigma_\ell \wedge \mathrm{opt} :: \mathrm{opt} :: *$ $[] \mapsto$ error $e :: es \mapsto \Sigma_\ell \wedge e :: e :: es$
pop	case $\Sigma_\ell^{\mathrm{old}}$ of $* \mapsto \Sigma_{\ell+1}$ $[] \mapsto$ error $e :: es \mapsto \Sigma_{\ell+1} \wedge es$	case $\Sigma_{\ell+1}^{\mathrm{old}}$ of $es \mapsto \Sigma_\ell \wedge \mathrm{opt} :: es$
dup	case $\Sigma_\ell^{\mathrm{old}}$ of $* \mapsto \Sigma_{\ell+1} \wedge \mathrm{opt} :: \mathrm{opt} :: *$ $[] \mapsto$ error $e :: es \mapsto \Sigma_{\ell+1} \wedge \mathrm{opt} :: e :: es$	case $\Sigma_{\ell+1}^{\mathrm{old}}$ of $* \mapsto \Sigma_\ell \wedge \mathrm{opt} :: *$ $[] \mapsto$ error $[e] \mapsto$ error $e_0 :: e_1 :: es \mapsto$ $\Sigma_\ell \wedge (e_0 \wedge e_1) :: es$
goto m	$\Sigma_m := \Sigma_m \wedge \Sigma_\ell^{\mathrm{old}}$	$\Sigma_\ell \wedge \Sigma_m^{\mathrm{old}}$
gotoF m	case $\Sigma_\ell^{\mathrm{old}}$ of $* \mapsto \Sigma_{\ell+1}$ $[] \mapsto$ error $e :: es \mapsto \Sigma_{\ell+1} \wedge es$ $\Sigma_m := \Sigma_m \wedge es$	case $\Sigma_{\ell+1}^{\mathrm{old}} \wedge \Sigma_m^{\mathrm{old}}$ of $es \mapsto \Sigma_\ell \wedge \mathrm{mnd} :: es$

The type-indexed similarity relation on states for establishing soundness of the optimization is defined as follows:

$$\frac{}{[] \sim_{[]} []} \qquad \frac{zs \sim_{es} zs_*}{z :: zs \sim_{\mathrm{mnd}::es} z :: zs_*} \qquad \frac{zs \sim_{es} zs_*}{z :: zs \sim_{\mathrm{opt}::es} zs_*} \qquad \frac{}{zs \sim_* []}$$

$$\frac{zs \sim_{es} zs_*}{(\ell, zs, \sigma) \sim_{es} (\ell, zs_*, \sigma)}$$

The rules state that two states are related, if they agree everywhere except for the optional stack positions in the first state, which must be omitted in the second.

The soundness statement is the same as in Theorem 26 and the proof is analogous. The same will hold for the optimizations in the next subsections.

Proof transformation Let $P|_\Sigma$ be P where all P_ℓ have been modified wrt. Σ_ℓ. Let $P_\ell|_{\Sigma_\ell} =_{\mathrm{df}} (\exists[v(st(n)) \mid es(n) = \mathrm{opt}]P_\ell[v(st(n))/st(n) \mid es(n) = \mathrm{opt}])[st(m-c)/st(m) \mid c = \mathsf{stoffset}(m, es)]$ where $es = \Sigma_\ell$, v is some assignment of unique logic variable names to stack positions and $\mathsf{stoffset}$ is defined as

$$\begin{array}{rcl} \mathsf{stoffset}(m, []) & = & 0 \\ \mathsf{stoffset}(0, \mathrm{mnd} :: es) & = & 0 \\ \mathsf{stoffset}(m, \mathrm{mnd} :: es) & = & \mathsf{stoffset}(m-1, es) \\ \mathsf{stoffset}(m, \mathrm{opt} :: es) & = & 1 + \mathsf{stoffset}(m, es). \end{array}$$

Informally, $P|_\Sigma$ is obtained from P by quantifying out stack positions which are opt, i.e., are not present in the optimized program and any stack position below the removed one is shifted up. For example, for assertion $st(0) = 4 \wedge st(1) = 5$ and type $[\mathrm{opt}, \mathrm{mnd}]$, the modified assertion becomes $\exists st0. st0 = 4 \wedge st(0) = 5$, which is equivalent to $st(0) = 5$.

We will use the following lemma.

Lemma 12 *(i) If $st(0) \notin P$ then $\mathsf{unshift}(P)|_{es} \equiv P|_{\mathrm{opt}::es}$ and (ii) $\mathsf{shift}(P)|_{\mathrm{opt}::es} \equiv P|_{es}$.*

We obtain the following theorem from proof transformation.

Theorem 28

1. *If $\Sigma \vdash (\ell, instr) \hookrightarrow (\ell, instr')$ and $P \vdash (\ell, instr)$ then $P|_\Sigma \vdash (\ell, instr')$*
2. *If $\Sigma \vdash c \hookrightarrow c'$ and $P \vdash c$, then also $P|_\Sigma \vdash c'$.*

Proof. For the constructive proof, we examine each typing rule. The interesting cases are the optimizing cases of load, push, pop and add.

- Case load$_2$. The typing judgement is of the following form

$$\frac{\mathrm{opt} :: \Sigma_\ell = \Sigma_{\ell+1}}{\Sigma \vdash (\ell, \mathsf{load}\ x) \hookrightarrow (\ell, \mathsf{nop})}$$

We also have the logic judgement of the form

$$\frac{P_\ell \models \mathsf{unshift}(P_{\ell+1}[x/st(0)])}{P \vdash (\ell, \mathsf{load}\ x)}$$

We have to show that

$$\frac{P_\ell|_{\Sigma_\ell} \models P_{\ell+1}|_{\Sigma_{\ell+1}}}{P \vdash (\ell, \mathsf{nop})}$$

We have the following sequence of entailments:

$$\begin{array}{lll}
P_\ell|_{\Sigma_\ell} & \models \; \mathsf{unshift}(P_{\ell+1}[x/st(0)])|_{\Sigma_\ell} & \text{by given entailment for load,} \\
& & \text{existential introduction and} \\
& & \text{elimination and variable renaming} \\
& \models \; P_{\ell+1}[x/st(0)]|_{\Sigma_{\ell+1}} & \text{by Lemma 12 and opt} :: \Sigma_\ell = \Sigma_{\ell+1} \\
& \models \; P_{\ell+1}|_{\Sigma_{\ell+1}} & \text{by existential introduction } (x \\
& & \text{is the witness for } v(st(0))
\end{array}$$

- Case push$_2$. Analogous to load.
- Case pop$_2$. The typing judgement is of the following form

$$\frac{\Sigma_\ell = \text{opt} :: \Sigma_{\ell+1}}{\Sigma \vdash (\ell, \mathsf{pop}) \hookrightarrow (\ell, \mathsf{nop})}$$

We also have the logic judgement of the form

$$\frac{P_\ell \models \mathsf{shift}(P_{\ell+1})}{P \vdash (\ell, \mathsf{pop})} \text{ pop}$$

We have to show that

$$\frac{P_\ell|_{\Sigma_\ell} \models P_{\ell+1}|_{\Sigma_{\ell+1}}}{P \vdash (\ell, \mathsf{nop})}$$

We have the following sequence of entailments

$$\begin{array}{lll}
P_{\ell+1}|_{\Sigma_{\ell+1}} & \models \; \mathsf{shift}(P_{\ell+1})|_{\Sigma_\ell} & \text{by the given entailment for pop,} \\
& & \text{existential introduction and} \\
& & \text{elimination and variable renaming} \\
& \models \; P_{\ell+1}|_{\Sigma_{\ell+1}} & \text{by Lemma 12}
\end{array}$$

- Case add$_2$. The typing judgement is of the form

$$\frac{\Sigma_l = \text{opt} :: \text{opt} :: es \quad \text{opt} :: es = \Sigma_{\ell+1}}{\Sigma \vdash (\ell, \mathsf{add}) \hookrightarrow (\ell, \mathsf{nop})} \text{ add}_2$$

We also have the logic judgement of the form

$$\frac{P_\ell \models \mathsf{shift}(P_{\ell+1})[st(0) + st(1)/st(1)]}{P \vdash (\ell, \mathsf{add})} \text{ add}$$

We have to show that

$$\frac{P_\ell|_{\Sigma_\ell} \models P_{\ell+1}|_{\Sigma_{\ell+1}}}{P \vdash (\ell, \mathsf{nop})}$$

We have the following sequence of entailments.

$$
\begin{array}{lll}
P_\ell|_{\Sigma_\ell} & \models & \mathsf{shift}(P_{\ell+1})[st(0)+st(1)/st(1)]|_{\Sigma_\ell} \\
& & \text{by the given entailment for add,} \\
& & \text{by existential introduction and elimination and variable renaming} \\
& \models & \mathsf{shift}(P_{\ell+1})|_{\Sigma_\ell} \quad st(0) \text{ and } st(1) \text{ opt in } \Sigma_\ell \\
& \models & P_{\ell+1}|_{\Sigma_{\ell+1}} \qquad \text{by Lemma 12 and } \Sigma_\ell = \text{opt} :: \Sigma_{\ell+1}.
\end{array}
$$

□

The following example illustrates the assertion transformation.

ℓ	Σ_ℓ	c_ℓ	c'_ℓ	P_ℓ	$P_\ell\|_{\Sigma_\ell}$
1	[]	load y	load y	$\top$	$\top$
2	[mnd]	load x	<u>nop</u>	$y = st(0)$	$y = st(0)$
3	[opt, mnd]	pop	<u>nop</u>	$y = st(1) \wedge st(0) = x$	$\exists s0.(y = st(0) \wedge s0 = x)$
4	[mnd]	store z	store z	$y = st(0)$	$y = st(0)$
5	[]			$y = z$	$y = z$

5.3 Store/load+ elimination

In this section, we deal with one of the more widely used bytecode optimizations—redundant store/load computations. This optimization is based on the observation that if a store is followed by a reload of the same variable to the same stack position (and the variable is not redefined in the meantime), then, provided there are no future uses of that variable, both the store and the load instruction can be eliminated. Similarly, if there is a store followed by n loads, then the store and loads can be replaced by $n-1$ dup instructions. Note that for these optimizations, the store and the loads do not necessarily have to be next to each other, there can be intervening instructions, as long as the stack height remains the same after the instructions and the values below the top are not consumed by them.

We approach this optimization in two stages. First, a simple forward copy propagation analysis determines whether some load instructions can be replaced with dup instructions. In the second stage, store/load pairs are detected and transformed.

5.3.1 Duplicating loads elimination

This analysis is a simple copy propagation analysis, which tries to determine if before a load x instruction, the value of x is already on top of the stack. If this is the case, the load x instruction can be replaced with a dup instruction. It is a unidirectional, forward analysis.

In the type system, label types are stack types or a special type "no state": **LabelType** $=_{\mathrm{df}}$ **StackType** $+ \{\emptyset\}$. Stack types $es \in$ **StackType** are lists of location types, which this time are elements of $x \in$ **Var** (signifying that the location

$$\frac{}{x \le \text{nac}} \qquad \frac{}{e \le e} \qquad \frac{}{[] \le []} \qquad \frac{e \le e' \quad es \le es'}{e :: es \le e' :: es'} \qquad \frac{}{\emptyset \le \tau} \qquad \frac{\forall \ell \in \mathbf{Label}. \Sigma_\ell \le \Sigma'_\ell}{\Sigma \le \Sigma'}$$

$$\frac{\Sigma_\ell = e :: es \quad \text{replace}(x, \text{nac}, es) \le \Sigma_{\ell+1}}{\Sigma \vdash (\ell, \mathsf{store}\ x) \hookrightarrow (\ell, \mathsf{store}\ x)}\ \text{store}$$

$$\frac{x :: \Sigma_\ell \le \Sigma_{\ell+1} \quad \forall es. \Sigma_l \ne x :: es}{\Sigma \vdash (\ell, \mathsf{load}\ x) \hookrightarrow (\ell, \mathsf{load}\ x)}\ \text{load}_1 \qquad \frac{x :: \Sigma_\ell \le \Sigma_{\ell+1} \quad \Sigma_l = x :: es}{\Sigma \vdash (\ell, \mathsf{load}\ x) \hookrightarrow (\ell, \mathsf{dup})}\ \text{load}_2$$

$$\frac{\text{nac} :: \Sigma_\ell \le \Sigma_{\ell+1}}{\Sigma \vdash (\ell, \mathsf{push}\ n) \hookrightarrow (\ell, \mathsf{push}\ n)}\ \text{push} \qquad \frac{\Sigma_\ell = e_0 :: e_1 :: es \quad \text{nac} :: es \le \Sigma_{\ell+1}}{\Sigma \vdash (\ell, \mathsf{add}) \hookrightarrow (\ell, \mathsf{add})}\ \text{add}$$

$$\frac{\Sigma_\ell = e :: es \quad es \le \Sigma_{\ell+1}}{\Sigma \vdash (\ell, \mathsf{pop}) \hookrightarrow (\ell, \mathsf{pop})}\ \text{pop} \qquad \frac{\Sigma_\ell = e :: es \quad e :: e :: es \le \Sigma_{\ell+1}}{\Sigma \vdash (\ell, \mathsf{dup}) \hookrightarrow (\ell, \mathsf{dup})}\ \text{dup}$$

$$\frac{\Sigma_\ell \le \Sigma_m}{\Sigma \vdash (\ell, \mathsf{goto}\ m) \hookrightarrow (\ell, \mathsf{goto}\ n)}\ \text{goto} \qquad \frac{\Sigma_\ell = e :: es \quad es \le \Sigma_{\ell+1} \quad es \le \Sigma_m}{\Sigma \vdash (\ell, \mathsf{gotoF}\ m) \hookrightarrow (\ell, \mathsf{gotoF}\ n)}\ \text{gotoF}$$

$$\frac{\Sigma_\ell = \emptyset}{\Sigma \vdash (\ell, instr) \hookrightarrow (\ell, instr)}\ \text{instr}$$

$$\frac{\forall \ell \in \text{dom}(c).\, \Sigma \vdash (\ell, c_\ell) \hookrightarrow (\ell, c'_\ell)}{\Sigma \vdash c \hookrightarrow c'}\ \text{code}$$

Figure 5.4: Type system for duplicating loads elimination

is certainly a copy of the variable x) and a special type "not a copy" (the location is possibly not a copy of any variable): $\mathbf{StackType} =_{\text{df}} \mathbf{LocType}^*$, $\mathbf{LabelType} =_{\text{df}} \mathbf{StackType} + \{\emptyset\}$ and $\mathbf{LocType} =_{\text{df}} \mathbf{Var} + \{\text{nac}\}$.

The subtyping and typing rules are given in Figure 5.4. The typing rules state that a stack position can be marked with a variable at label ℓ if on all paths to this label, this variable is put on the stack in this position, and not modified later. In other words, at label ℓ the value in the corresponding position in the stack is necessarily equal to the value of the variable. If a stack type holds a nac element in some position it means that this position may not be a copy (e.g., since on some path to ℓ, a numeral is pushed to that position).

Thus the typing rule for load reflects that, after the instruction, the value on top of the stack and the value of the corresponding variable are necessarily equal. A $\mathsf{store}\ x$ explicitly kills all variables x in the stack, since the values in the stack and the new value of the variable cannot be guaranteed to be consistent anymore. The function replace means exactly that in the store rule. An optimization can be made, if a variable x is on top of the stack before a $\mathsf{load}\ x$ instruction. In this case, the load can be replaced with a dup, as in the following example:

Σ_ℓ	(ℓ, c_ℓ)	(ℓ, c'_ℓ)
$[]$	$0, \mathsf{load}\ x$	$0, \mathsf{load}\ x$
$[x]$	$1, \mathsf{push}\ 1$	$1, \mathsf{push}\ 1$
$[\text{nac}, x]$	$2, \mathsf{store}\ y$	$2, \mathsf{store}\ y$
$[x]$	$3, \mathsf{load}\ x$	$3, \mathsf{dup}$
$[]$	$4,$	$4,$

This optimization could be improved by not only keeping track of copies of variables in the stack, but also in the variables. A location type would then be a set of variables (those variables of which the given location is certainly a copy; the empty set will signify that the location is possibly not a copy of anything). Then, even if there were consecutive loads from different variables, a dup could be introduced, provided that the two variables were actually copies of each other.

The principal type inference algorithm calculates the least type of a piece of code c greater than a given code type Σ_{init} (with $\Sigma_\ell^{\text{init}} = \bot$ for most ℓ), if such exists, iterating a forward update procedure. The general form of the algorithm is as Algorithm 1, only forward updating.

The (partial) lub operations are the following:

lubs of value types			
$x \vee x$	$=_{\text{df}}$	x	
$x \vee x'$	$=_{\text{df}}$	nac	if $x \neq x'$
$\text{nac} \vee e$	$=_{\text{df}}$	nac	
$e \vee \text{nac}$	$=_{\text{df}}$	nac	

lubs of label types		
$\bot$	$=_{\text{df}}$	$\emptyset$
$\emptyset \vee es$	$=_{\text{df}}$	es
$es \vee \emptyset$	$=_{\text{df}}$	es
$[] \vee []$	$=_{\text{df}}$	$[]$
$[] \vee (e :: es)$	$=_{\text{df}}$	error
$(e :: es) \vee []$	$=_{\text{df}}$	error
$(e :: es) \vee (e' :: es')$	$=_{\text{df}}$	$(e \vee e') :: (es \vee es')$

The update procedures are as follows:

c_ℓ	$\mathrm{updfwd}_\ell, \Sigma_{\ell+1} :=$
store x	case $\Sigma_\ell^{\mathrm{old}}$ of $\emptyset \mapsto \Sigma_{\ell+1}$ $[] \mapsto$ error $e :: es \mapsto \Sigma_{\ell+1} \vee \mathrm{replace}(x, \mathrm{nac}, es)$
load x	case $\Sigma_\ell^{\mathrm{old}}$ of $\emptyset \mapsto \Sigma_{\ell+1}$ $es \mapsto \Sigma_{\ell+1} \vee x :: es$
push n	case $\Sigma_\ell^{\mathrm{old}}$ of $\emptyset \mapsto \Sigma_{\ell+1}$ $es \mapsto \Sigma_{\ell+1} \vee \mathrm{nac} :: es$
add	case $\Sigma_\ell^{\mathrm{old}}$ of $\emptyset \mapsto \Sigma_{\ell+1}$ $[] \mapsto$ error $[e] \mapsto$ error $e_0 :: e_1 :: es \mapsto \Sigma_{\ell+1} \vee \mathrm{nac} :: es$
pop	case $\Sigma_\ell^{\mathrm{old}}$ of $\emptyset \mapsto \Sigma_{\ell+1}$ $[] \mapsto$ error $e :: es \mapsto \Sigma_{\ell+1} \vee es$
dup	case $\Sigma_\ell^{\mathrm{old}}$ of $\emptyset \mapsto \Sigma_{\ell+1}$ $[] \mapsto$ error $e :: es \mapsto \Sigma_{\ell+1} \vee e :: e :: es$
goto m	$\Sigma_m := \Sigma_m \vee \Sigma_\ell^{\mathrm{old}}$
gotoF m	case $\Sigma_\ell^{\mathrm{old}}$ of $\emptyset \mapsto \Sigma_{\ell+1}$ $[] \mapsto$ error $e :: es \mapsto \Sigma_{\ell+1} \vee es;$ $\Sigma_m := \Sigma_m \vee es$

The label-type indexed similarity relation on states to establish soundness of the optimization is defined as follows:

$$\frac{}{[] \sim_{[]}^{\sigma} []} \qquad \frac{zs \sim_{es}^{\sigma} zs_*}{z :: zs \sim_{\mathrm{nac}::es}^{\sigma} z :: zs_*} \qquad \frac{zs \sim_{es}^{\sigma} zs_*}{\sigma(x) :: zs \sim_{x::es}^{\sigma} \sigma(x) :: zs_*} \qquad \frac{zs \sim_{es}^{\sigma} zs_*}{(\ell, zs, \sigma) \sim_{es} (\ell, zs_*, \sigma)}$$

(Note that no states are in the relation $\sim_\emptyset$.)

Proof transformation Let $P|_\Sigma$ be P where all P_ℓ have been modified wrt. Σ_ℓ. Let $P_\ell|_{\Sigma_\ell} =_{\mathrm{df}} P_\ell \wedge \bigwedge[st(n) = x \mid es(n) = x]$ where $es = \Sigma_\ell$. Informally, $P_\ell|_{\Sigma_\ell}$ is the conjunction of P_ℓ and the equalities of corresponding stack positions and variables in type Σ_ℓ.

Lemma 13 $\mathsf{unshift}(P)|_{es} \equiv \mathsf{unshift}(P|_{\mathrm{nac}::es})$

We obtain the following theorem for proof transformation.

Theorem 29

1. *If* $\Sigma \vdash (\ell, instr) \hookrightarrow (\ell, instr')$ *and* $P \vdash (\ell, instr)$ *then* $P|_{\Sigma} \vdash (\ell, instr')$.

2. *If* $\Sigma \vdash c \hookrightarrow c'$ *and* $P \vdash c$, *then also* $P|_{\Sigma} \vdash c'$.

Proof. The only interesting case is for load_2.

- Case load_2. The typing judgement is of the following form

$$\frac{x :: \Sigma_\ell \leq \Sigma_{\ell+1} \quad \Sigma_l = x :: es}{\Sigma \vdash (\ell, \mathsf{load}\ x) \hookrightarrow (\ell, \mathsf{dup})}\ \text{load}_2$$

  We also have the logic judgement of the form

$$\frac{P_\ell \models \mathsf{unshift}(P_{\ell+1}[x/st(0)])}{P \vdash (\ell, \mathsf{load}\ x)}$$

  We have to show that

$$\frac{P_\ell|_{\Sigma_\ell} \models \mathsf{unshift}(P_{\ell+1}|_{\Sigma_{\ell+1}}[st(1)/st(0)])}{P \vdash (\ell, \mathsf{dup})}$$

  We have the following sequence of entailments.

$$\begin{array}{llll}
P_\ell|_{\Sigma_\ell} & \models & \mathsf{unshift}(P_{\ell+1}[x/st(0)])|_{\Sigma_\ell} & \text{by the given entailment for load} \\
& \models & \mathsf{unshift}(P_{\ell+1}[x/st(0)])|_{x::es} & \Sigma_\ell = x :: es \\
& \models & \mathsf{unshift}(P_{\ell+1}[x/st(0)]|_{\text{nac}::x::es}) & \text{by Lemma 13} \\
& \models & \mathsf{unshift}(P_{\ell+1}|_{x::x::es}[x/st(0)]) & (st(0) = x)[x/st(0)] \equiv true \\
& \models & \mathsf{unshift}(P_{\ell+1}|_{x::x::es}[st(1)/st(0)]) & \text{by substitution of equals} \\
& & & \text{for equals, since } st(1) = x \\
& \models & \mathsf{unshift}(P_{\ell+1}|_{\Sigma_{\ell+1}}[st(1)/st(0)]) & x :: x :: es \leq \Sigma_{\ell+1}
\end{array}$$

$\square$

Example

Let us look at the previously shown example with precondition $x = 3$ and postcondition $st(0) = 3$. The transformation is the following.

Σ_ℓ	(ℓ, c_ℓ)	(ℓ, c'_ℓ)	P	$P\|_\Sigma$
$[]$	$0, \mathsf{load}\ x$	$0, \mathsf{load}\ x$	$x = 3$	$x = 3$
$[x]$	$1, \mathsf{push}\ 1$	$1, \mathsf{push}\ 1$	$x = 3$	$x = 3 \wedge st(0) = x$
$[\text{nac}, x]$	$2, \mathsf{store}\ y$	$2, \mathsf{store}\ y$	$x = 3$	$x = 3 \wedge st(1) = x$
$[x]$	$3, \mathsf{load}\ x$	$3, \mathsf{dup}$	$x = 3$	$x = 3 \wedge st(0) = x$
$[]$	$4,$	$4,$	$st(0) = 3$	$st(0) = 3$

5.3.2 Store-load pairs elimination

The store-load pairs analysis tries to find store instructions followed by a load instruction to the same stack position and referring to the same variable (before any new store to the same variable takes place). Provided that this variable is not used later on, both instructions could be eliminated. Since the possible future use of a variable requires a live variable analysis, we take the approach to only remove the load instruction, but keep the store instruction and precede it with a dup, as in the following example:

$$\begin{array}{ll} 0, \mathsf{store}\ x & 0, \mathsf{dup} \\ 1, \mathsf{load}\ x & 1, \mathsf{store}\ x \\ 2, \ldots & 2, \ldots \end{array}$$

The benefit of this approach is that a check for future uses of x can be omitted. If it turns out that the variable is not needed, a dead code elimination optimization would remove the dup and store instructions later on.

Since this optimization manipulates pairs of instructions, a bidirectional analysis is needed, as was the case with load-pop pairs.

Similarly to the previous analysis, label types are again stack types or the special type $\emptyset$ for "no state": $\mathbf{LabelType} =_{\mathrm{df}} \mathbf{StackType} + \{\emptyset\}$. Stack types $es \in \mathbf{StackType}$ are lists of location types, which are elements x of $\mathbf{Var}$ (a position to be inserted in the optimized code to keep a copy of variable x in the stack) and "mandatory" (original positions): $\mathbf{StackType} =_{\mathrm{df}} \mathbf{LocType}^*$, $\mathbf{LocType} =_{\mathrm{df}} \mathbf{Var} + \{\mathrm{mnd}\}$.

The subtyping and typing rules are given in Figure 5.5. The typing rules say that, if a label has some stack type, then every time this label is reached in the execution of the code the size of the stack will be equal to the number of mnd elements in the stack. In addition, if at some label the stack type contains an element of **Var**, it means that, if the code were optimized according to the typing rules, then at that label in the optimized code, the stack would hold an additional copy of that variable between the positions corresponding to the positions of the original code.

The optimization rules for the analysis are presented in Figure 5.6. The optimization is slightly more complex than the previous ones. Since a single instruction might be transformed into 2 instructions, a remapping of labels is potentially needed. We use the auxiliary function offset which returns the offset of an instruction (the difference between its position in the optimized program and the original program), taking into account the optimizations happening at lower labels. The function takes as its argument the original position (label) and the type context.

As an example, a piece of code could be typed in the following way:

$$\frac{}{[] \leq []} \qquad \frac{es \leq es'}{e :: es \leq e :: es'} \qquad \frac{es \leq es'}{x :: es \leq es'} \qquad \frac{}{\emptyset \leq \tau} \qquad \frac{\forall \ell \in \mathbf{Label}. \Sigma_\ell \leq \Sigma'_\ell}{\Sigma \leq \Sigma'}$$

$$\frac{\Sigma_\ell = \mathrm{mnd} :: es \quad es = \Sigma_{\ell+1} \quad \neg\mathrm{member}(x, es)}{\Sigma \vdash (\ell, \mathsf{store}\ x)} \ \mathrm{store}_1$$

$$\frac{\Sigma_\ell = \mathrm{mnd} :: es \quad x :: es = \Sigma_{\ell+1} \quad \neg\mathrm{member}(x, es)}{\Sigma \vdash (\ell, \mathsf{store}\ x)} \ \mathrm{store}_2$$

$$\frac{\Sigma_\ell = es \quad \mathrm{mnd} :: es = \Sigma_{\ell+1}}{\Sigma \vdash (\ell, \mathsf{load}\ x)} \ \mathrm{load}_1 \qquad \frac{\Sigma_\ell = x : es \quad \mathrm{mnd} :: es = \Sigma_{\ell+1}}{\Sigma \vdash (\ell, \mathsf{load}\ x)} \ \mathrm{load}_2$$

$$\frac{\mathrm{mnd} :: \Sigma_\ell = \Sigma_{\ell+1}}{\Sigma \vdash (\ell, \mathsf{push}\ n)} \ \mathrm{push} \qquad \frac{\Sigma_\ell = \mathrm{mnd} :: \mathrm{mnd} :: es \quad \Sigma_{\ell+1} = \mathrm{mnd} :: es}{\Sigma \vdash (\ell, \mathsf{add})} \ \mathrm{add}$$

$$\frac{\Sigma_\ell = \mathrm{mnd} :: \Sigma_{\ell+1}}{\Sigma \vdash (\ell, \mathsf{pop})} \ \mathrm{pop} \qquad \frac{\Sigma_\ell = \mathrm{mnd} :: es \quad \mathrm{mnd} :: \mathrm{mnd} :: es = \Sigma_{\ell+1}}{\Sigma \vdash (\ell, \mathsf{dup})} \ \mathrm{dup}$$

$$\frac{\Sigma_\ell = \Sigma_m}{\Sigma \vdash (\ell, \mathsf{goto}\ m)} \ \mathrm{goto} \qquad \frac{\Sigma_\ell = \mathrm{mnd} :: es \quad es = \Sigma_{\ell+1} \quad es = \Sigma_m}{\Sigma \vdash (\ell, \mathsf{gotoF}\ m)} \ \mathrm{gotoF}_1$$

$$\frac{\Sigma_\ell = \emptyset \quad \emptyset = \Sigma_{\ell+1}}{\Sigma \vdash (\ell, instr)} \ \mathrm{nonjump} \qquad \frac{\Sigma_\ell = \emptyset \quad \emptyset = \Sigma_{\ell+1} \quad \emptyset = \Sigma_m}{\Sigma \vdash (\ell, \mathsf{gotoF}\ m)} \ \mathrm{gotoF}_2$$

$$\frac{\forall \ell \in \mathrm{dom}(c).\ \Sigma \vdash (\ell, c_\ell)}{\Sigma \vdash c} \ \mathrm{code}$$

Figure 5.5: Type system for store-load pairs elimination

$$\frac{\Sigma_\ell = \mathrm{mnd} :: es \quad es = \Sigma_{\ell+1} \quad \neg\mathrm{member}(x, es) \quad n = \mathsf{offset}(\ell, \Sigma)}{\Sigma \vdash (\ell, \mathsf{store}\ x) \hookrightarrow \{(\ell + n, \mathsf{store}\ x)\}} \ \mathrm{store}_1$$

$$\frac{\Sigma_\ell = \mathrm{mnd} :: es \quad x :: es = \Sigma_{\ell+1} \quad \neg\mathrm{member}(x, es) \quad n = \mathsf{offset}(\ell, \Sigma)}{\Sigma \vdash (\ell, \mathsf{store}\ x) \hookrightarrow \{(\ell + n, \mathsf{dup}), (\ell + n + 1, \mathsf{store}\ x)\}} \ \mathrm{store}_2$$

$$\frac{\Sigma_\ell = es \quad \mathrm{mnd} :: es = \Sigma_{\ell+1} \quad n = \mathsf{offset}(\ell, \Sigma)}{\Sigma \vdash (\ell, \mathsf{load}\ x) \hookrightarrow \{(\ell + n, \mathsf{load}\ x)\}} \ \mathrm{load}_1 \qquad \frac{\Sigma_\ell = x : es \quad \mathrm{mnd} :: es = \Sigma_{\ell+1} \quad n = \mathsf{offset}(\ell, \Sigma)}{\Sigma \vdash (\ell, \mathsf{load}\ x) \hookrightarrow \emptyset} \ \mathrm{load}_2$$

$$\frac{\mathrm{mnd} :: \Sigma_\ell = \Sigma_{\ell+1} \quad n = \mathsf{offset}(\ell, \Sigma)}{\Sigma \vdash (\ell, \mathsf{push}\ n) \hookrightarrow \{(\ell + n, \mathsf{push}\ n)\}} \ \mathrm{push} \qquad \frac{\Sigma_\ell = \mathrm{mnd} :: \mathrm{mnd} :: es \quad \Sigma_{\ell+1} = \mathrm{mnd} :: es \quad n = \mathsf{offset}(\ell, \Sigma)}{\Sigma \vdash (\ell, \mathsf{add}) \hookrightarrow \{(\ell + n, \mathsf{add})\}} \ \mathrm{add}$$

$$\frac{\Sigma_\ell = \mathrm{mnd} :: \Sigma_{\ell+1} \quad n = \mathsf{offset}(\ell, \Sigma)}{\Sigma \vdash (\ell, \mathsf{pop}) \hookrightarrow \{(\ell + n, \mathsf{pop})\}} \ \mathrm{pop} \qquad \frac{\Sigma_\ell = \mathrm{mnd} :: es \quad \mathrm{mnd} :: \mathrm{mnd} :: es = \Sigma_{\ell+1} \quad n = \mathsf{offset}(\ell, \Sigma)}{\Sigma \vdash (\ell, \mathsf{dup}) \hookrightarrow \{(\ell + n, \mathsf{dup})\}} \ \mathrm{dup}$$

$$\frac{\Sigma_\ell = \Sigma_m \quad \Sigma_m = es \quad n_1 = \mathsf{offset}(\ell, \Sigma) \quad n_2 = \mathsf{offset}(m, \Sigma)}{\Sigma \vdash (\ell, \mathsf{goto}\ m) \hookrightarrow \{(\ell + n_1, \mathsf{goto}\ m + n_2)\}} \ \mathrm{goto}$$

$$\frac{\Sigma_\ell = \mathrm{mnd} :: es \quad es = \Sigma_{\ell+1} \quad es = \Sigma_m \quad n_1 = \mathsf{offset}(\ell, \Sigma) \quad n_2 = \mathsf{offset}(m, \Sigma)}{\Sigma \vdash (\ell, \mathsf{gotoF}\ m) \hookrightarrow \{(\ell + n_1, \mathsf{gotoF}\ m + n_2)\}} \ \mathrm{gotoF}_1$$

$$\frac{\Sigma_\ell = \emptyset \quad \emptyset = \Sigma_{\ell+1} \quad n = \mathsf{offset}(\ell, \Sigma)}{\Sigma \vdash (\ell, instr) \hookrightarrow \{(\ell + n, instr)\}} \ \mathrm{nonjump} \qquad \frac{\Sigma_\ell = \emptyset \quad \emptyset = \Sigma_{\ell+1} \quad \emptyset = \Sigma_m \quad n = \mathsf{offset}(\ell, \Sigma)}{\Sigma \vdash (\ell, \mathsf{gotoF}\ m) \hookrightarrow \{(\ell + n, \mathsf{gotoF}\ m)\}} \ \mathrm{gotoF}_2$$

$$\frac{\forall \ell \in \mathrm{dom}(c). \Sigma \vdash (\ell, c_\ell) \hookrightarrow z_\ell}{\Sigma \vdash c \hookrightarrow \bigcup\{z_\ell \mid \ell \in \mathrm{dom}(c)\}} \ \mathrm{code}$$

Figure 5.6: Optimizing type system for store-load pairs elimination

Σ_ℓ	ℓ, c_ℓ	ℓ, c'_ℓ
$[]$	$0, \mathsf{push}\ 1$	$0, \mathsf{push}\ 1$
$[\mathrm{mnd}]$	$1, \mathsf{store}\ x$	$1, \mathsf{dup}$
$[x]$	$2, \mathsf{push}\ 1$	$2, \mathsf{store}\ x$
$[\mathrm{mnd}, x]$	$3, \mathsf{store}\ y$	$3, \mathsf{push}\ 1$
$[x]$	$4, \mathsf{load}\ x$	$4, \mathsf{store}\ y$
$[\mathrm{mnd}]$	$5, \mathsf{store}\ z$	$5, \mathsf{dup}$
$[z]$	$6,$	$6, \mathsf{store}\ z$

Here, the offset of instructions 2 and 3 is one. The offset of instructions 1 and 5 is zero (the offset is the net change of the labels *preceding* the given label.

Note that the type given is the principal type that the analysis algorithm would derive when the type of label 6 is initialized to $\emptyset$. It is also acceptable to type label 6 with $[]$.

The analysis algorithm works as follows. The lowest acceptable types of some labels should be given (typically the type of the entry point of the program, i.e., label 0, is set to $[]$ and optionally also the types of the exit labels). The rest of the labels are initialized to be of the default type $\emptyset$. The algorithm then tries to compute the actual and potential stack heights for each label. As mentioned before, the mnd elements in the type mean the actual elements; the variables in the type mean potential elements present after optimization. For store and load, "speculative" types can initially be given, since a store could potentially be replaced with a dup and a store (so that after the transformation an extra value would be on the stack) and load with nop (since the extra value already present removes the need of reloading it). The types of the successors and predecessors of a label are used to compute the local label type. The types arriving from different directions are combined by a non-deterministic union operation determined by the subtyping relation. (Both $[x]$ and $[y]$ are greater than $[x, y]$ and $[y, x]$, so there is no unique least upper bound for them, only minimal upper bounds.) As a result, any provisional additional position is dropped from the stack, if it turns out that on some incoming or outgoing path the potential added stack height is not really viable. At first sight, the nondeterministic lub may seem counterintuitive, but is in fact natural for this kind of optimization, since there can be two different equally strong optimizations for a given piece of code. This is can be illustrated on the following example.

Σ^1_ℓ	Σ^2_ℓ	(ℓ, c_ℓ)	(ℓ, c^1_ℓ)	(ℓ, c^2_ℓ)
$[]$	$[]$	$0, \mathsf{push}\ 1$	$0, \mathsf{push}\ 1$	$0, \mathsf{push}\ 1$
$[\mathrm{mnd}]$	$[\mathrm{mnd}]$	$1, \mathsf{store}\ x$	$1, \mathsf{dup}$	$1, \mathsf{store}\ x$
$[x]$	$[]$	$2, \mathsf{push}\ 2$	$2, \mathsf{store}\ x$	$2, \mathsf{push}\ 2$
$[\mathrm{mnd}, x]$	$[\mathrm{mnd}]$	$3, \mathsf{store}\ y$	$3, \mathsf{push}\ 2$	$3, \mathsf{dup}$
$[x]$	$[y]$	$4, \mathsf{load}\ x$	$4, \mathsf{store}\ y$	$4, \mathsf{store}\ y$
$[\mathrm{mnd}]$	$[\mathrm{mnd}, y]$	$5, \mathsf{store}\ z$	$5, \mathsf{store}\ z$	$5, \mathsf{load}\ x$
$[]$	$[y]$	$6, \mathsf{load}\ y$	$6, \mathsf{load}\ y$	$6, \mathsf{store}\ z$
$[\mathrm{mnd}]$	$[\mathrm{mnd}]$	$7, \mathsf{store}\ w$	$7, \mathsf{store}\ w$	$7, \mathsf{store}\ w$
$[]$	$[]$	$8,$	$8,$	$8,$

In this example, load from either x or y can be removed by inserting a dup instructions before either a store to x or y, respectively. The two possible typings for code c are Σ^1 and Σ^2, which result in optimized programs c^1 and c^2. It is not possible to remove both load instructions at the same time, at least not without rearranging them.

The type inference algorithm calculates nondeterministically a minimal type of a piece of code c greater than a given code type Σ_{init} (with $\Sigma^{\mathrm{init}}_\ell = \bot$ for most ℓ), if such a type exists, iterating forward and backward update procedures. The general form of the algorithm is the same as for Algorithm 2.

The (partial, non-deterministic) minimal upper bound operations are the following:

$$
\begin{array}{rcl}
\multicolumn{3}{c}{\text{mubs of label types}} \\
\hline
\bot & =_{\mathrm{df}} & \emptyset \\
\emptyset \vee es & =_{\mathrm{df}} & es \\
es \vee \emptyset & =_{\mathrm{df}} & es \\
[] \vee [] & =_{\mathrm{df}} & [] \\
[] \vee x :: es & =_{\mathrm{df}} & [] \vee es \\
[] \vee \mathrm{mnd} :: es & =_{\mathrm{df}} & \mathsf{error} \\
x :: es \vee [] & =_{\mathrm{df}} & es \vee [] \\
\mathrm{mnd} :: es \vee [] & =_{\mathrm{df}} & \mathsf{error} \\
x :: es \vee x :: es' & =_{\mathrm{df}} & x :: (es \vee es') \\
x :: es \vee x' :: es' & =_{\mathrm{df}} & es \vee x' :: es' \quad \text{if } x \neq x' \\
x :: es \vee x' :: es' & =_{\mathrm{df}} & x :: es \vee es' \quad \text{if } x \neq x' \\
x :: es \vee \mathrm{mnd} :: es' & =_{\mathrm{df}} & es \vee (\mathrm{mnd} :: es') \\
\mathrm{mnd} :: es \vee x :: es' & =_{\mathrm{df}} & (\mathrm{mnd} :: es) \vee es' \\
\mathrm{mnd} :: es \vee \mathrm{mnd} :: es' & =_{\mathrm{df}} & \mathrm{mnd} :: (es \vee es')
\end{array}
$$

The update procedures are the following:

c_ℓ	$\mathrm{updfwd}_\ell, \Sigma_{\ell+1} :=$
store x	case $\Sigma_\ell^{\mathrm{old}}$ of $\emptyset \mapsto \Sigma_{\ell+1}$ $xs \mapsto$ error $xs{+\!\!+}\mathrm{mnd} :: es \mapsto \Sigma_{\ell+1} \vee x :: \mathrm{remove}(x, es)$
load x	case $\Sigma_\ell^{\mathrm{old}}$ of $\emptyset \mapsto \Sigma_{\ell+1}$ $x :: es \mapsto \Sigma_{\ell+1} \vee \mathrm{mnd} :: es$ $es \mapsto \Sigma_{\ell+1} \vee \mathrm{mnd} :: es$
push n	case $\Sigma_\ell^{\mathrm{old}}$ of $\emptyset \mapsto \Sigma_{\ell+1}$ $es \mapsto \Sigma_{\ell+1} \vee \mathrm{mnd} :: es$
add	case $\Sigma_\ell^{\mathrm{old}}$ of $\emptyset \mapsto \Sigma_{\ell+1}$ $xs_0 \mapsto$ error $xs_0{+\!\!+}(\mathrm{mnd} :: xs_1) \mapsto$ error $xs_0{+\!\!+}(\mathrm{mnd} :: xs_1){+\!\!+}(\mathrm{mnd} :: es) \mapsto \Sigma_{\ell+1} \vee \mathrm{mnd} :: es$
pop	case $\Sigma_\ell^{\mathrm{old}}$ of $\emptyset \mapsto \Sigma_{\ell+1}$ $xs \mapsto$ error $xs{+\!\!+}(\mathrm{mnd} :: es) \mapsto \Sigma_{\ell+1} \vee es$
dup	case $\Sigma_\ell^{\mathrm{old}}$ of $\emptyset \mapsto \Sigma_{\ell+1}$ $xs \mapsto$ error $xs{+\!\!+}(\mathrm{mnd} :: es) \mapsto \Sigma_{\ell+1} \vee \mathrm{mnd} :: \mathrm{mnd} :: es$
goto m	$\Sigma_m := \Sigma_m \vee \Sigma_\ell^{\mathrm{old}}$
gotoF m	case $\Sigma_\ell^{\mathrm{old}}$ of $\emptyset \mapsto \Sigma_{\ell+1}$ $xs \mapsto$ error $xs{+\!\!+}(\mathrm{mnd} :: es) \mapsto \Sigma_{\ell+1} \vee es;$ $\Sigma_m := \Sigma_m \vee es$

c_ℓ	$\text{updbwd}_\ell, \Sigma_\ell :=$
store x	case $\Sigma^{\text{old}}_{\ell+1}$ of $\emptyset \mapsto \Sigma_\ell$ $x :: es \mapsto \Sigma_\ell \vee \text{mnd} :: es$ $es \mapsto \Sigma_\ell \vee \text{mnd} :: es$
load x	case $\Sigma^{\text{old}}_{\ell+1}$ of $\emptyset \mapsto \Sigma_\ell$ $xs \mapsto$ error $xs +\!\!+ (\text{mnd} :: es) \mapsto \Sigma_\ell \vee x :: es$
push n	case $\Sigma^{\text{old}}_{\ell+1}$ of $\emptyset \mapsto \Sigma_\ell$ $xs \mapsto$ error $xs +\!\!+ (\text{mnd} :: es) \mapsto \Sigma_\ell \vee es$
add	case $\Sigma^{\text{old}}_{\ell+1}$ of $\emptyset \mapsto \Sigma_\ell$ $xs \mapsto$ error $xs +\!\!+ (\text{mnd} :: es) \mapsto \Sigma_\ell \vee \text{mnd} :: \text{mnd} :: es$
pop	case $\Sigma^{\text{old}}_{\ell+1}$ of $\emptyset \mapsto \Sigma_\ell$ $es \mapsto \Sigma_\ell \vee \text{mnd} :: es$
dup	case $\Sigma^{\text{old}}_{\ell+1}$ of $\emptyset \mapsto \Sigma_\ell$ $xs_0 \mapsto$ error $xs_0 +\!\!+ (\text{mnd} :: xs_1) \mapsto$ error $xs_0 +\!\!+ (\text{mnd} :: xs_1) +\!\!+ (\text{mnd} :: es) \mapsto \Sigma_\ell \vee \text{mnd} :: es$
goto m	$\Sigma_\ell := \Sigma_\ell \vee \Sigma^{\text{old}}_m$
gotoF m	case $\Sigma^{\text{old}}_{\ell+1} \vee \Sigma^{\text{old}}_m$ of $\emptyset \mapsto \Sigma_\ell$ $es \mapsto \Sigma_\ell \vee \text{mnd} :: es$

The similarity relation between states wrt. to a given type is defined as follows:

$$\frac{}{[] \sim^\sigma_{[]} []} \qquad \frac{\ell' = \ell + \mathsf{offset}(\ell, \Sigma)}{\ell \sim_\Sigma \ell'}$$

$$\frac{zs \sim^\sigma_{es} zs_*}{z :: zs \sim^\sigma_{\text{mnd}::es} z :: zs_*} \qquad \frac{zs \sim^\sigma_{es} zs_*}{zs \sim^\sigma_{x::es} \sigma(x) :: zs_*} \qquad \frac{zs \sim^\sigma_{es} zs_* \quad \ell \sim_\Sigma \ell'}{(\ell, zs, \sigma) \sim_\Sigma (\ell', zs_*, \sigma)}$$

Since the optimization can change instruction labels, it needs to be accounted for in the similarity relation. Two states are related, if the second one has appropriate additional positions in its stack component, agreeing suitably with the store component (additional positions copy variables) and the labels are shifted according to the type. Again no two states are in the relation $\sim_\emptyset$.

Proof transformation Let $P_\ell|_{es} = P_\ell[st(m + c)/st(m) \mid c = \mathsf{stoffset}(m, es)] \wedge \bigwedge[st(n) = x \mid es(n) = x]$, where stoffset is defined as

$$\begin{array}{rcl} \mathsf{stoffset}(m, []) & = & 0 \\ \mathsf{stoffset}(0, \mathrm{mnd} :: es) & = & 0 \\ \mathsf{stoffset}(m, \mathrm{mnd} :: es) & = & \mathsf{stoffset}(m-1, es) \\ \mathsf{stoffset}(m, x :: es) & = & 1 + \mathsf{stoffset}(m, es) \end{array}$$

Intuitively, $P_\ell|_{es}$ is obtained from P_ℓ by shifting all original stack positions in the assertion according to the type ("making room" for the new stack positions), and all stack positions containing a variable are made equal to that variable in the assertion. For example for assertion $st(0) = 5 \wedge st(1) = 7$ and type $x :: \mathrm{mnd} :: y :: \mathrm{mnd}$, we get the new assertion $st(1) = 5 \wedge st(3) = 7 \wedge st(0) = x \wedge st(2) = y$.

When defining the modified P wrt. Σ, i.e. $P|_\Sigma$, we have to take into account that the size and the labeling of the code can change during optimizations so the domain of the assertion vector also changes. The new assertion vector $P|_\Sigma$ is defined as

$$P|_\Sigma(n) = \begin{cases} P_\ell|_{\Sigma_\ell} & \text{if } n = \ell + \mathsf{offset}(\ell, \Sigma) \\ \mathsf{shift}(P_\ell|_{\Sigma_\ell}) \wedge st(0) = st(1) & \text{if } \Sigma_{\ell+1} = x :: es \text{ and } n = \ell + \mathsf{offset}(\ell, \Sigma) + 1 \end{cases}$$

This maps all the (possibly label shifted) unoptimized instructions to their new labels, while the optimized store instructions get a new assertions (second case in the equation).

The following example shows the original and modified assertions. The assertions at labels 2, 3 and 6 are shifted, while assertions for store instructions at label 1 and 5 are expanded into 2 new assertions.

Σ_ℓ	ℓ, c_ℓ	P_ℓ	ℓ, c'_ℓ	$P\|_{\Sigma}{}_\ell$
$[]$	0, push 1	$1 = 1$	0, push 1	$1 = 1$
$[\mathrm{mnd}]$	1, store x	$st(0) = 1 \wedge 2 = 2$	1, dup	$st(0) = 1 \wedge 2 = 2$
$[x]$	2, push 2	$x = 1 \wedge 2 = 2$	2, store x	$st(1) = 1 \wedge 2 = 2 \wedge st(0) = st(1)$
$[\mathrm{mnd}, x]$	3, store y	$x = 1 \wedge st(0) = 2$	3, push 2	$x = 1 \wedge 2 = 2 \wedge st(0) = x$
$[x]$	4, load x	$x = 1 \wedge y = 2 \wedge x = x$	4, store y	$x = 1 \wedge st(0) = 2 \wedge st(1) = x$
$[\mathrm{mnd}]$	5, store z	$x = 1 \wedge y = 2 \wedge st(0) = x$	5, dup	$x = 1 \wedge y = 2 \wedge st(0) = x$
$[z]$	6,	$x = 1 \wedge y = 2 \wedge z = x$	6, store z	$x = 1 \wedge y = 2 \wedge st(0) = x \wedge st(0) = st(1)$
	7,		7,	$x = 1 \wedge y = 2 \wedge z = x \wedge st(0) = z$

Lemma 14 $\mathsf{shift}(P)|_{\mathrm{mnd}::es} \equiv \mathsf{shift}(P|_{es})$.

We obtain the following theorem for proof transformation.

Theorem 30

1. *If* $\Sigma \vdash (\ell, instr) \hookrightarrow z$ *and* $P \vdash (\ell, instr)$ *then* $P|_\Sigma \vdash z$
2. *If* $\Sigma \vdash c \hookrightarrow c'$ *and* $P \vdash c$, *then also* $P|_\Sigma \vdash c'$.

Proof. The interesting case is the store x optimization.

- Case store_2. The typing judgement is of the following form

$$\frac{\Sigma_\ell = \text{mnd} :: es \quad x :: es = \Sigma_{\ell+1} \quad \neg\text{member}(x, es) \quad n = \mathsf{offset}(\ell, \Sigma)}{\Sigma \vdash (\ell, \mathsf{store}\ x) \hookrightarrow \{(\ell+n, \mathsf{dup}), (\ell+n+1, \mathsf{store}\ x)\}}$$

We also have the logic judgement of the form

$$\frac{P_\ell \models \mathsf{shift}(P_{\ell+1})[st(0)/x]}{P \vdash (\ell, \mathsf{store}\ x)}\ \text{store}$$

We have to show that

$$\frac{P|_\Sigma(\ell+n) \models \mathsf{unshift}(P|_\Sigma(\ell+n+1)[st(1)/st(0)])}{P|_\Sigma \vdash (\ell+n, \mathsf{dup})}\ \text{dup}$$

and

$$\frac{P|_\Sigma(\ell+n+1) \models \mathsf{shift}(P|_\Sigma(\ell+n+2))[st(0)/x]}{P|_\Sigma \vdash (\ell+n+1, \mathsf{store}\ x)}\ \text{store}$$

We know that $P|_\Sigma(\ell+n) = P_\ell|_{\Sigma_\ell} = P_\ell|_{\text{mnd}::es}$ and since $\Sigma_{\ell+1} = x :: es$, then $P|_\Sigma(\ell+n+1) = \mathsf{shift}(P_\ell)|_{\Sigma_\ell} \wedge st(0) = st(1) = \mathsf{shift}(P_\ell)|_{\text{mnd}::es} \wedge st(0) = st(1)$ and $P|_\Sigma(\ell+n+2) = P_{\ell+1}|_{\Sigma_{\ell+1}} = P_{\ell+1}|_{x::es}$.

  - For the dup rule, we need to show that $P_\ell|_{\text{mnd}::es} \models \mathsf{unshift}((\mathsf{shift}(P_\ell|_{\text{mnd}::es}) \wedge st(0) = st(1))[st(1)/st(0)])$. We have the following sequence of entailments.

$$\begin{array}{ll}
P_\ell|_{\text{mnd}::es} \models & \\
\quad \models & \mathsf{unshift}(\mathsf{shift}(P_\ell|_{\text{mnd}::es})) \\
\quad \models & \mathsf{unshift}(\mathsf{shift}(P_\ell|_{\text{mnd}::es})[st(1)/st(0)]) \\
 & \qquad\qquad\qquad\qquad \mathsf{shift}(P) \text{ does not contain } st(0) \\
\quad \models & \mathsf{unshift}(\mathsf{shift}(P_\ell|_{\text{mnd}::es})[st(1)/st(0)] \wedge st(1) = st(1)) \\
\quad \models & \mathsf{unshift}((\mathsf{shift}(P_\ell|_{\text{mnd}::es}) \wedge st(0) = st(1))[st(1)/st(0)])
\end{array}$$

  - For store rule, we have to show that $(\mathsf{shift}(P_\ell|_{\text{mnd}::es}) \wedge st(0) = st(1)) \models \mathsf{shift}(P_{\ell+1}|_{x::es})[st(0)/x]$.

    We know that $P_\ell \models \mathsf{shift}(P_{\ell+1})[st(0)/x]$. We have the following sequence of entailments.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathsf{shift}(P_\ell|_{\mathrm{mnd}:es}) \wedge st(0) = st(1)} \\
\models & \mathsf{shift}(P_\ell)|_{\mathrm{mnd::mnd::}es} \wedge st(0) = st(1) \qquad \text{by Lemma 14} \\
\models & \mathsf{shift}(\mathsf{shift}(P_{\ell+1})[st(0)/x])|_{\mathrm{mnd::mnd:}es} \wedge st(0) = st(1) \\
 & \qquad \text{By the given entailment for store, variable renaming} \\
 & \qquad \text{and conjunction introduction and elimination} \\
\models & \mathsf{shift}(\mathsf{shift}(P_{\ell+1})[st(0)/x]|_{\mathrm{mnd:}es}) \wedge st(0) = st(1) \\
 & \qquad \text{by Lemma 14} \\
\models & (\mathsf{shift}(\mathsf{shift}(P_{\ell+1})|_{\mathrm{mnd:}es}[st(0)/x]) \wedge st(0) = st(1) \\
 & \qquad \text{by } \neg\text{member}(x, es) \\
\models & \mathsf{shift}(\mathsf{shift}(P_{\ell+1}|_{es})[st(0)/x]) \wedge st(0) = st(1) \\
 & \qquad \text{by Lemma 14} \\
\models & \mathsf{shift}(\mathsf{shift}(P_{\ell+1}|_{es}))[st(0)/x] \wedge st(0) = st(1) \\
 & \qquad \text{since } st(0) = st(1) \\
\models & \mathsf{shift}(P_{\ell+1}|_{x::es})[st(0)/x] \wedge st(0) = st(1) \\
 & \qquad x :: es \text{ accounts for the } \mathsf{shift}() \\
\models & \mathsf{shift}(P_{\ell+1}|_{x::es})[st(0)/x]
\end{array}
$$

□

Example

As an example lets look at a program $s \equiv \mathsf{while}\ x < n\ \mathsf{do}\ x := x + 1; y := x * x; z := z + y$. The corresponding bytecode program c, its type Σ and its optimized version c' are the following.

ℓ	Σ	c_ℓ	c'_ℓ
1	$[]$	load n	load n
2	$[\text{mnd}]$	load x	load x
3	$[\text{mnd}, \text{mnd}]$	less	less
4	$[\text{mnd}]$	gotoF 18	gotoF 18
5	$[]$	push 1	push 1
6	$[\text{mnd}]$	load x	load x
7	$[\text{mnd}, \text{mnd}]$	add	add
8	$[\text{mnd}]$	store x	$\underline{\text{dup}}$
9	$[x]$	load x	$\underline{\text{store } x}$
10	$[\text{mnd}]$	load x	load x
11	$[\text{mnd}, \text{mnd}]$	mult	mult
12	$[\text{mnd}]$	store y	$\underline{\text{dup}}$
13	$[y]$	load y	$\underline{\text{store } y}$
14	$[\text{mnd}]$	load z	load z
15	$[\text{mnd}, \text{mnd}]$	add	add
16	$[\text{mnd}]$	store z	store z
17	$[]$	goto 1	goto 1
18			

Suitable pre- and postconditions for the program are $x = 0 \wedge z = 0 \wedge n \geq 0$ and $z = \sum_{i=1}^{n} i^2$, respectively. The suitable assertions for each label are

Assertion	ℓ	Instruction
$x = 0 \wedge z = 0 \wedge n > 0$		
$x < n \wedge x \leq n \wedge z + (x+1) * (x+1) = ... \vee x \not< n \wedge ...$	1	load n
$x < st(0) \wedge x \leq n \wedge z + (x+1) * (x+1) = ... \vee x \not< st(0) \wedge ...$	2	load x
$st(0) < st(1) \wedge x \leq n \wedge z + (x+1) * (x+1) = ... \vee st(0) \not< st(1) \wedge ...$	3	less
$st(0) \wedge x \leq n \wedge z + (x+1) * (x+1) = ... \vee \neg st(0) \wedge ...$	4	gotoF 18
$x \leq n \wedge z + (x+1) * (x+1) = \sum_{i=1}^{x+1} i^2$	5	push 1
$x \leq n \wedge z + (x + st(0)) * (x + st(0)) = \sum_{i=1}^{x+st(0)} i^2$	6	load x
$x \leq n \wedge z + (st(0) + st(1)) * (st(0) + st(1)) = \sum_{i=1}^{st(0)+st(1)} i^2$	7	add
$x \leq n \wedge z + st(0) * st(0) = \sum_{i=1}^{st(0)} i^2$	8	store x
$x \leq n \wedge z + x * x = \sum_{i=1}^{x} i^2$	9	load x
$x \leq n \wedge z + x * st(0) = \sum_{i=1}^{x} i^2$	10	load x
$x \leq n \wedge z + st(0) * st(1) = \sum_{i=1}^{x} i^2$	11	mult
$x \leq n \wedge z + st(0) = \sum_{i=1}^{x} i^2$	12	store y
$x \leq n \wedge z + y = \sum_{i=1}^{x} i^2$	13	load y
$x \leq n \wedge z + st(0) = \sum_{i=1}^{x} i^2$	14	load z
$x \leq n \wedge st(0) + st(1) = \sum_{i=1}^{x} i^2$	15	add
$x \leq n \wedge st(0) = \sum_{i=1}^{x} i^2$	16	store z
$x \leq n \wedge z = \sum_{i=1}^{x} i^2$	17	goto 1
$z = \sum_{i=1}^{n} i^2$	18	

The assertions for the optimized program are

$x = 0 \wedge z = 0 \wedge n > 0$		
$x < n \wedge x \leq n \wedge z + (x+1) * (x+1) = ... \vee x \not< n \wedge ...$	1	load n
$x < st(0) \wedge x \leq n \wedge z + (x+1) * (x+1) = ... \vee x \not< st(0) \wedge ...$	2	load x
$st(0) < st(1) \wedge x \leq n \wedge z + (x+1) * (x+1) = ... \vee st(0) \not< st(1) \wedge ...$	3	less
$st(0) \wedge x \leq n \wedge z + (x+1) * (x+1) = ... \vee \neg st(0) \wedge ...$	4	gotoF 18
$x \leq n \wedge z + (x+1) * (x+1) = \sum_{i=1}^{x+1} i^2$	5	push 1
$x \leq n \wedge z + (x + st(0)) * (x + st(0)) = \sum_{i=1}^{x+st(0)} i^2$	6	load x
$x \leq n \wedge z + (st(0) + st(1) * (st(0) + st(1))) = \sum_{i=1}^{st(0)+st(1)} i^2$	7	add
$x \leq n \wedge z + st(0) * st(0) = \sum_{i=1}^{st(0)} i^2$	8	dup
$x \leq n \wedge z + st(1) * st(1) = \sum_{i=1}^{st(0)} i^2 \wedge st(0) = st(1)$	9	store x
$x \leq n \wedge z + x * st(0) = \sum_{i=1}^{x} i^2 \wedge x = st(0)$	10	dup
$x \leq n \wedge z + st(0) * st(1) = \sum_{i=1}^{x} i^2$	11	mult
$x \leq n \wedge z + st(0) = \sum_{i=1}^{x} i^2$	12	dup
$x \leq n \wedge z + st(1) = \sum_{i=1}^{x} i^2 \wedge st(0) = st(1)$	13	store y
$x \leq n \wedge z + st(0) = \sum_{i=1}^{x} i^2$	14	load z
$x \leq n \wedge st(0) + st(1) = \sum_{i=1}^{x} i^2$	15	add
$x \leq n \wedge st(0) = \sum_{i=1}^{x} i^2$	16	store z
$x \leq n \wedge z = \sum_{i=1}^{x} i^2$	17	goto 1
$z = \sum_{i=1}^{n} i^2$		

5.4 Related work

As mentioned in the introduction, one of the more well-known bytecode transformation tools is Soot [64]. Soot's approach to bytecode optimization is to transform bytecode into 3-address intermediate code, to use standard techniques to optimize the intermediate code, and then to translate the code back into bytecode. The back and forth translation can introduce several inefficiencies into bytecode, such as redundant store/load computations. This is tackled by either transforming the intermediate code into an aggregated form, using some peephole optimization techniques and converting it to bytecode using standard tree traversal techniques, or by translating the intermediate code into a streamlined form of bytecode, and performing store-load optimizations on its basic blocks [65]. The benefit of Soot's approach is of course that optimizations on the intermediate language are routine to perform. The drawback is that multiple transformations between different representations make the optimizations performed non-transparent and the code can lose some properties that were present before. This can become an issue, when preservation of properties beyond the standard semantics (e.g., code size) is desired.

The Java bytecode analyzer Julia [60] provides a framework for implementing different static analyses on Java bytecode. Analyses implemented so far include escape analysis, rapid type analysis, information-flow analysis, static initialisation analysis and several others. To our knowledge the analyses outlined here have not been implemented in Julia.

The jDFA framework performs basic analysis of liveness and constant propagation on bytecode [43]. Liveness information is only computed for local registers (not for the stack). More complicated analyses, which would allow removal of dead code

or constant folding, are not implemented. Further implementation of this framework seems to have stopped.

VanDrunen et al. [67] give a formalization and soundness proofs for specific pattern based eliminations of store-load pairs in basic blocks. Their work is partly motivated by that of Shpeisman and Tikir [59], who list specific instances for code replacement in Java bytecode, considering variations of dup instruction available there, but do not formalize these transformations.

Our analyses and optimizations are presented in the form of type systems, following the approach of Stata and Abadi [61], who described the Java bytecode verifier as a type system. Bidirectional analyses for high-level languages have been described in great detail by Khedker and Dhamdhere [32]. Such analyses have been used in high-level optimizations such as the original Morel-Renvoise algorithm [44] for partial redundancy elimination and several subsequent ones [24, 22] (discussed more thoroughly in the previous chapter).

5.5 Conclusion

We have described four different data-flow analyses for optimizing stack-based code. Our main goal was to give general and formal descriptions for both the analyses and optimizations based on them. We have outlined the difficulties associated to bytecode optimizations that modify pairs of stack height changing instructions across basic block boundaries. The analyses have been presented in the form of type systems, which lend themselves for proof transformation, being in this respect very similar to the high-level optimizing type systems presented in the previous chapter.

We have also presented the algorithms for computing strongest analyses. The analyses have been implemented for full Java bytecode, except for Jsr/Ret instructions (which have been deprecated in Java's SDK since version 6 [68]). The implementations follow closely the algorithms presented, using the ASM [4] bytecode manipulation framework. Additionally, we have implemented the Simple PRE algorithm (described in the previous chapter) for Java bytecode. In this case we do some preprocessing on bytecode, to discover expressions which can potentially be moved. The expression moving phase, which is trivial for a high-level language (simply by replacing an expression on the right hand side of an assignment with a temporary variable) is more complicated for bytecode since expressions can potentially span several basic blocks. To overcome this, we use the dead-store and load-pop elimination optimizations. For example to remove an expression, it suffices to "mark" it with a pop instruction, and then run the load-pop elimination on the code.

Indeed, while the algorithms presented in this chapter are not particularly performance-enhancing on their own in general, they can be thought of as building blocks for other, more complicated optimization. For high-level language the optimization/code-rewriting part is in general easy after the required dataflow analyses are performed, but this is not necessarily the case for bytecode, since moving around code poses its own challenges. One interesting line of work would be to

show how this can be overcome by chaining together the "utility" analyses with optimization-specific analyses like anticipability in case of Simple PRE.

Chapter 6

Conclusions and future work

This thesis concentrated on two aspects relevant to PCC. We demonstrated that introducing compositional big-step semantics and Hoare logics for languages with jumps is quite straightforward by using finite unions of pieces of code as the underlying phrase structure. We did this on the example of a bytecode-like language. The semantic and logic descriptions are fundamentally no more complicated than those for structured high-level languages. Especially noteworthy is that Hoare triples in our logic are interpreted in the standard way, unlike other logics targeting similar languages [15, 63, 62]. Quite naturally, Hoare triples of While can be compiled into Hoare triples of the bytecode language.

We also proposed a viable solution for dealing with proof "optimization". The main insight there was to describe dataflow analyses as type systems. Since the shape of the type derivation tree matches the Hoare derivation tree, it is easy to transform assertions in places relevant to the optimization, where the transformation is guided by the particular type. This is in contrast to transforming proof obligations generated by weakest precondition calculus. In the latter case, there is no explicit connection between the program point that is modified in the transformation and to the part of the proof that needs to be modified accordingly, the connection is only indirect. This makes proof transformation using the type-systematic approach much more straightforward.

Future work There are several paths along which this work can be extended.

In this thesis, we only dealt with classical Hoare logic (without procedures) and intraprocedural optimizations. The work on proof transformation should be extended to *interprocedural optimizations* (for example interprocedural constant propagation or common subexpression elimination). This would assume working with a logic that handles procedure/method calls. It would also be interesting to look at optimizations for object-oriented programs, e.g. static initialization optimization, elimination of dynamic casts, etc. Another line of future work in the same vein would be considering transformations based on alias analysis for program proofs in separation logic. We believe the way to go would still be by using type-indexed similarity relations.

Some of the paths we have already mentioned. It would be of interest to investigate how similarity relations could be *systematically* derived from the type system. Right now, we craft the similarity relation in an ad hoc way. It should be possible to build a systematic framework for extracting the similarity from the type system (both for analysis and optimization type systems).

It would also be of interest to look into *modularizing soundness proofs* of optimizations as explained in Chapter 4. In this thesis, if an optimization was a result of a cascade of analyses, we proved the optimization sound in one monolithic step. Instead, each analysis could be proven correct separately, using different instrumented semantics along the way. Then, those proofs could be reused for proving sound a similar optimization which uses some of the same analyses. As we already argued before, the drawback of the modular proof is that, more often than not, dataflow analyses abstract not over states, but over some other properties of computation, such as computation paths and expression evaluations on them. This means that in order to prove the analysis sound, non-standard instrumented semantics need to be introduced, which are in general different for each individual analysis. The benefit of the monolithic proof is that it is based only on standard (big-step) semantics, so no new formalism needs to be introduced.

In Chapter 5, we described two *bidirectional* analyses, where information is propagated both back and fourth in the control flow graph. A notable feature of the type system for bidirectional analysis is that it does not have the usual subsumption rule (for weakening/strengthening the pre/posttype), since changing a type in one program point is not possible in the general case: any change might influence types in other program points. This means that subsumption is only applicable for a "web" of types that are related to each other. An interesting line of work would be to look more deeply into the theoretical aspects of type systems for bidirectional analysis. We have already made some progress in this area.

Bibliography

[1] E. Albert, G. Puebla, and M. V. Hermenegildo. Abstraction-carrying code. In F. Baader and A. Voronkov, editors, *Proc. of 11th Int. Conf. Logic for Programming Artificial Intelligence and Reasoning, LPAR 2004*, volume 3452 of *Lecture Notes in Computer Science*, pages 380–397. Springer-Verlag, 2005.

[2] A. Appel and D. McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Transactions on Programming Languages and Systems*, 23(5):657–683, 2001.

[3] M. A. Arbib and S. Alagić. Proof rules for **goto**s. *Acta Informatica*, 11:139–148, 1979.

[4] ASM. `http://asm.objectweb.org/`, 2008 (accessed September 20, 2008).

[5] D. Aspinall, L. Beringer, and A. Momigliano. Optimisation validation. In *Proc. of 5th Int. Wksh. on Compiler Optimization Meets Compiler Verification, COCV 2006*, volume 13 of *Electronic Notes in Theoretical Computer Science*, pages 573–600. Elsevier, 2006.

[6] F. Y. Bannwart and P. Müller. A program logic for bytecode. In F. Spoto, editor, *Proc. of Bytecode Semantics, Verification, Analysis and Transformation*, volume 141 of *Electronic Notes in Theoretical Computer Science*, pages 255–273. Elsevier, 2005.

[7] G. Barthe, L. Burdy, J. Charles, B. Grégoire, M. Huisman, J.-L. Lanet, M. Pavlova, and A. Requet. JACK: a tool for validation of security and behaviour of Java applications. In F. S. de Boer et al., editors, *Proc. of 5th Int. Symp. on Formal Methods for Components and Objects, FMCO 2006*, volume 4709 of *Lecture Notes in Computer Science*, pages 152–174. Springer-Verlag, 2007.

[8] G. Barthe, B. Grégoire, C. Kunz, and T. Rezk. Certificate translation for optimizing compilers. In K. Yi, editor, *Proc. of 13th Int. Static Analysis Symposium, SAS 2006*, volume 4134 of *Lecture Notes in Computer Science*, pages 301–317. Springer-Verlag, 2006.

[9] G. Barthe and C. Kunz. Certificate translation in abstract interpretation. In S. Drossopoulou, editor, *Proc. of 17th European Symposium on Programming, ESOP 2008*, volume 4960 of *Lecture Notes in Computer Science*, pages 368–382. Springer-Verlag, 2008.

[10] G. Barthe and T. Rezk. Non-interference for a JVM-like language. In M. Fähndrich and G. Morrisett, editors, *Proc. of 2005 ACM SIGPLAN Int. Wksh. on Types in Language Design and Implementation, TLDI 2005*, pages 103–112. ACM Press, 2005.

[11] G. Barthe, T. Rezk, and A. Saabas. Proof obligations preserving compilation. In T. Dimitrakos, F. Martinelli, P. Ryan, and S. Schneider, editors, *Proc. of 3rd Int. Wksh. on Formal Aspects in Security and Trust, FAST 2005*, volume 3866 of *Lecture Notes in Computer Science*, pages 112–126. Springer-Verlag, 2005.

[12] B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *Lecture Notes in Computer Science*. Springer-Verlag, 2007.

[13] N. Benton. Simple relational correctness proofs for static analyses and program transformations. In N. D. Jones and X. Leroy, editors, *Proc. of 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004*, pages 14–25. ACM Press, 2004.

[14] N. Benton. A typed logic for stacks and jumps, 2004. Draft.

[15] N. Benton. A typed, compositional logic for a stack-based abstract machine. In K. Yi, editor, *Proc. of 3rd Asian Symp. on Programming Languages and Systems, APLAS 2005*, volume 3780 of *Lecture Notes in Computer Science*, pages 364–380. Springer-Verlag, 2005.

[16] Y. Bertot, B. Grégoire, and X. Leroy. A structured approach to proving compiler optimizations based on dataflow analysis. In J.-C. Filliâtre, C. Paulin-Mohring, and B. Werner, editors, *Revised Selected Papers from 1st Int. Wksh. on Types for Proofs and Programs, TYPES 2004*, number 3839 in Lecture Notes in Computer Science, pages 66–81. Springer-Verlag, 2004.

[17] D. Bronnikov. A practical adoption of partial redundancy elimination. *ACM SIGPLAN Notices*, 39(8):49–53, 2004.

[18] P. Chalin, J. R. Kiniry, G. T. Leavens, and E. Poll. Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In F. S. de Boer et al., editors, *Proc. of 5th Int. Symp. on Formal Methods for Components and Objects, FMCO 2006*, volume 4111 of *Lecture Notes in Computer Science*, pages 342–363. Springer-Verlag, 2006.

[19] S. A. Cook. Soundness and completeness of an axiom system for verification. *SIAM Journal on Computing*, 7(1):70–90, 1978.

[20] A. de Bruin. Goto statements: Semantics and deduction systems. *Acta Informatica*, 15:385–424, 1981.

[21] M. Debbabi, A. Mourad, C. Talhi, and H. Yahyaoui. Accelerating embedded Java for mobile devices. *IEEE Communications*, 43(9):80–85, 2005.

[22] D. M. Dhamdhere. Practical adaptation of the global optimization algorithm of Morel and Renvoise. *ACM Transactions on Programming Languages and Systems*, 13(2):291–294, 1991.

[23] D. M. Dhamdhere. E-path_PRE—partial redundancy elimination made easy. *ACM SIGPLAN Notices*, 37(8):53–65, 2002.

[24] K. H. Drechsler and M. P. Stadel. A solution to a problem with Morel and Renvoise's "Global optimization by suppression of partial redundancies". *ACM Transactions on Programming Languages and Systems*, 10(4):635–640, 1988.

[25] K. H. Drechsler and M. P. Stadel. A variation of Knoop, Rüthing and Steffen's "Lazy code motion". *ACM SIGPLAN Notices*, 28(5):29–38, 1993.

[26] K. H. Drechsler and M. P. Stadel. Optimal code motion: theory and practice. *ACM Transactions on Programming Languages and Systems*, 16(4):1117–1155, 1994.

[27] J.-C. Filliâtre. Why: a multi-language multi-prover verification tool. Research Report 1366, LRI, Université Paris Sud, 2003.

[28] J.-C. Filliâtre and C. Marché. Multi-prover verification of C programs. In J. Davies, W. Schulte, and M. Barnett, editors, *Proc. of 6th International Conference on Formal Engineering Methods, ICFEM 2004*, volume 3308 of *Lecture Notes in Computer Science*, pages 15–29. Springer-Verlag, 2000.

[29] R. W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume 19 of *Proc. of Symp. in Appl. Math.*, pages 19–33. American Mathematical Society, 1967.

[30] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[31] M. Huisman and B. Jacobs. Java program verification via a Hoare logic with abrupt termination. In T. Maibaum, editor, *Proc. of Fundamental Approaches to Software Engineering*, volume 1783 of *Lecture Notes in Computer Science*, pages 284–303. Springer-Verlag, 2000.

[32] U. P. Khedker and D. M. Dhamdhere. A generalized theory of bit vector data flow analysis. *ACM Transactions on Programming Languages and Systems*, 16(5):1472–1511, 1994.

[33] J. Knoop, O. Rüthing, and B. Steffen. Lazy code motion. In *Proc. of the ACM SIGPLAN'92 Conf. on Programming Language Design and Implementation, PLDI 1992*, pages 224–234. ACM Press, 1992.

[34] N. Kobayashi and K. Kirane. Type-based information analysis for low-level languages. In *Proc. of 3rd Asian Wksh. on Programming Languages and Systems, APLAS'02*, pages 302–316, 2003.

[35] P. J. Koopman. A preliminary exploration of optimized stack code generation. *Journal of Forth Applications and Research*, 6(3):241–251, 1994.

[36] T. Kowaltowski. Axiomatic approach to side effects and general jumps. *Acta Informatica*, 7:357–360, 1977.

[37] D. Lacey and E. Van Wyk N. D. Jones. Proving correctness of compiler optimizations by temporal logic. *Higher-Order and Symbolic Computation*, 17(3):173–206, 2004.

[38] P. Laud, T. Uustalu, and V. Vene. Type systems equivalent to data-flow analyses for imperative languages. *Theoretical Computer Science*, 364(3):292–310, 2006.

[39] S. Lerner, T. Millstein, and C. Chambers. Automatically proving the correctness of compiler optimizations. In *Proc. of the ACM SIGPLAN 2003 Conf. on Programming Language Design and Implementation, PLDI 2003*, pages 220–231. ACM Press, 2003.

[40] S. Lerner, T. Millstein, E. Rice, and C. Chambers. Automatically proving the correctness of compiler optimizations. In J. Palsberg and M. Abadi, editors, *Proc. of 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005*, pages 364–377. ACM Press, 2005.

[41] M. Maierhofer and M. A. Ertl. Local stack allocation. In K. Koskimies, editor, *Proc. of 7th Int. Conf. on Compiler Construction, CC '98*, volume 1383 of *Lecture Notes in Computer Science*, pages 189–203. Springer-Verlag, 1998.

[42] C. Marché, C. Paulin-Mohring, and X. Urbain. The Krakatoa tool for certification of Java/JavaCard programs annotated with JML annotations. *Journal of Logic and Algebraic Programming*, 58:89–106, 2004.

[43] M. Mohnen. An open framework for data-flow analysis in Java. In *Proc. of 2nd Wksh. on Intermediate Representation Engineering for Virtual Machines*, 2002.

[44] E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22(2):96–103, 1979.

[45] G. C. Necula. Proof-carrying code. In *Proc. of 24st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1997*, pages 106–119. ACM Press, 1997.

[46] G. C. Necula. Translation validation for an optimizing compiler. In *Proc. of the ACM SIGPLAN 2000 Conf. on Programming Language Design and Implementation, PLDI 2000*, pages 88–94. ACM Press, 2000.

[47] G. C. Necula and P. Lee. Efficient representation and validation of proofs. In *Proc. of the 13th IEEE Symp. on Logic in Computer Science, LICS 1998*, pages 93–104. IEEE Press, 1998.

[48] H. R. Nielson and F. Nielson, editors. *Semantics with Applications: A Formal Introduction.* Wiley, 1992.

[49] H. R. Nielson and F. Nielson. Flow logic: a multi-paradigmatic approach to static analysis. In *The Essence of Computation, Complexity, Analysis, Transformation: Essays Dedicated to Neil D. Jones*, volume 2566 of *Lecture Notes in Computer Science*, pages 223–244. Springer-Verlag, 2002.

[50] V. K. Paleri, Y. N. Srikant, and P. Shankar. Partial redundancy elimination: a simple, pragmatic, and provably correct algorithm. *Science of Computer Programming*, 48(1):1–20, 2003.

[51] C. L. Quigley. A programming logic for Java bytecode programs. In D. A. Basin and B. Wolff, editors, *Proc of. 16th Int. Conf. on Theorem Proving in Higher-Order Logic, TPHOLs 2003*, volume 2758 of *Lecture Notes in Computer Science*, pages 41–54. Springer-Verlag, 2003.

[52] A. Saabas and T. Uustalu. Compositional type systems for stack-based low-level languages. In B. Jay and J. Gudmundssson, editors, *Proc. of 12th Computing: Australasian Theory Symp., CATS 2006*, volume 51 of *Confs. in Research and Practice in Inform. Techn.*, pages 27–39. Australian Computer Society, 2006.

[53] A. Saabas and T. Uustalu. A compositional natural semantics and Hoare logic for low-level languages. *Theoretical Computer Science*, 373(3):273–302, 2007.

[54] A. Saabas and T. Uustalu. Type systems for optimizing stack-based code. In M. Huisman and F. Spoto, editors, *Proc. of 2nd Wkshp. on Bytecode Semantics, Verification, Analysis and Transformation, BYTECODE 2007*, volume 190(1) of *Electron. Notes in Theor. Comput. Sci.*, pages 103–119. Elsevier, 2007.

[55] A. Saabas and T. Uustalu. Program and proof optimizations with type systems. *Journal of Logic and Algebraic Programming*, 77(1–2):131–154, 2008.

[56] A. Saabas and T. Uustalu. Proof optimization for partial redundancy elimination. In R. Glück and O. de Moor, editors, *Proc. of 2008 ACM SIGPLAN Wksh. on Partial Evaluation and Semantics-Based Program Manipulation, PEPM 2008*, pages 91–101. ACM Press, 2008.

[57] L. Schröder and T. Mossakowski. Monad-independent Hoare logic in HasCASL. In M. Pezzè, editor, *Proc. of 6th Int. Conf. on Fundamental Approaches to Software Engineering, FASE 2003*, volume 2621 of *Lecture Notes in Computer Science*, pages 261–277. Springer-Verlag, 2003.

[58] L. Schröder and T. Mossakowski. Generic exception handling and the Java monad. In C. Rattray, S. Maharaj, and C. Shankland, editors, *Proc. of 10th Int. Conf. on Algebraic Methodology and Software Technology, AMAST 2004*, number 3116 in Lecture Notes in Computer Science, pages 443–459. Springer-Verlag, 2004.

[59] T. Shpeisman and M. Tikir. Generating efficient stack code for Java. Technical Report CS-TR-4069, University of Maryland, 1999.

[60] F. Spoto. JULIA: A generic static analyser for the Java bytecode. In *Proc. of Workshop on Formal Techniques for Java Programs, FTfJP 2005*, 2005.

[61] R. Stata and M. Abadi. A type system for Java bytecode subroutines. *ACM Transactions on Programming Languages and Systems*, 21(1):90–137, 1999.

[62] G. Tan. *A compositional logic for control flow and its application for proof-carrying code*. PhD thesis, Princeton Univ, 2005.

[63] G. Tan and A. W. Appel. A compositional logic for control flow. In E. A. Emerson and K. S. Namjoshi, editors, *Proc. of 7th Int. Conf. on Verification, Model Checking, and Abstract Interpretation, VMCAI 2006*, volume 3855 of *Lecture Notes in Computer Science*, pages 80–94. Springer-Verlag, 2006.

[64] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot – a Java bytecode optimization framework. In *Proc. of Conf. on the Centre of Advanced Studies for Collaborative Research*, pages 125–135, 1999.

[65] R. Vallée-Rai, E. Gagnon, L. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Local stack allocation. In D. A. Watt, editor, *Proc. of 9th Int. Conf. on Compiler Construction, CC 2000*, volume 1781 of *Lecture Notes in Computer Science*, pages 18–34. Springer-Verlag, 2000.

[66] J. van den Berg and B. Jacobs. The LOOP compiler for Java and JML. In T. Margaria and W. Yi, editors, *Proc. of 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2001*, volume 2031 of *Lecture Notes in Computer Science*, pages 299–312. Springer-Verlag, 2001.

[67] T. VanDrunen, A. L. Hosking, and J. Palsberg. Reducing loads and stores in stack architectures, 2000. Draft.

[68] New JDK 6 Verifier. `https://jdk.dev.java.net/verifier.html`, 2008 (accessed September 20, 2008).

[69] J. Xue and J. Knoop. A fresh look at PRE as a maximum flow problem. In A. Mycroft and A. Zeller, editors, *Proc. of 15th Int. Conf. on Compiler Construction, CC 2006*, volume 3923 of *Lecture Notes in Computer Science*, pages 139–154. Springer-Verlag, 2006.

[70] L. Zuck, A. Pnueli, Y. Fang, and B. Goldberg. Voc: a methodology for the translation validation of optimizing compilers. *Journal of Universal Computer Science*, 9(3):223–247, 2003.

List of Publications

- M. Veanes, A. Saabas. On bounded reachability of programs with set comprehensions. Proceedings of the 15th International Conference on Logic for Programming, Artificial Intelligence and Reasoning, LPAR 2008 (Doha, Qatar, November 2008). Lecture Notes in Computer Science, Springer, 2008. To appear.

- A. Saabas, T. Uustalu. Program and proof optimizations with type systems. Journal of Logic and Algebraic Programming, v. 77, n. 1-2, pp. 131-154, 2008.

- A. Saabas, T. Uustalu. Proof optimization for partial redundancy elimination. Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM'08 (San Francisco, January 2008), pp. 91-101, ACM, 2008.

- M. J. Frade, A. Saabas, T. Uustalu. Foundational certification of data-flow analyses. Proceedings of the 1st IEEE and IFIP International Symposium on Theoretical Aspects of Software Engineering, TASE 2007 (Shanghai, June 2007), pp. 107-116. IEEE CS Press, 2007.

- A. Saabas, T. Uustalu. Type systems for optimizing stack-based code. Proceedings of the 2nd Workshop on Bytecode Semantics, Verification, Analysis and Transformation, Bytecode 2007 (Braga, March 2007), v. 190, n. 1 of Electron. Notes in Theoretical Computer Science, pp. 103-119. Elsevier, 2007.

- A. Saabas, T. Uustalu. A compositional natural semantics and Hoare logic for low-level languages. Theoretical Computer Science, v. 373, n. 3, pp. 273-302, 2007.

- A. Saabas, T. Uustalu. Compositional Type Systems for Stack Based Low-Level Languages. Proceedings of the 12th Computing, Australasian Theory Symp., CATS 2006 (Hobart, Jan. 2006), v. 51 of Confs. in Research and Practice in Inform. Techn., pp. 27-39. Australian Comput. Soc., 2006.

- A. Saabas, T. Uustalu. A compositional natural semantics and Hoare logic for low-level languages. Proceedings of the 2nd Workshop on Structured Operational Semantics, SOS 2005 (Lisbon, July 2005), v. 156, n. 1 of Electronic Notes in Theoretical Computer Science, pp. 151-168. Elsevier, 2006.

- P. Grigorenko, A. Saabas, E. Tyugu. Visual tool for generative programming. Proceedings of the 10th European Software Engineering Conference, ESEC/FSE-13 (Lisbon, Sept. 2005), pp. 249-252, ACM Press, 2005.

- G. Barthe, T. Rezk, A. Saabas. Proof Obligations Preserving Compilation. Proceedings of the 3rd Workshop on Formal Aspects in Security and Trust, 2005 (Newcastle upon Tyne, 2005), v. 3866 of Lecture Notes in computer Science, pp. 112-126, Springer Verlag, 2006
- P. Grigorenko, A. Saabas, E. Tyugu. COCOVILA - Compiler-Compiler for Visual Languages. Proceedings of the 5th Workshop on Language Descriptions, Tools and Applications, 2005 (Edinburgh, April 2005), v. 141, n. 4 of Electronic Notes in Theoretical Computer Science, pp. 137-142. Elsevier, 2005.

Curriculum Vitae

Personal Data

Name	Ando Saabas
Date of birth	21.09.1979
Citizenship	Estonian

Contact data

Address	Institute of Cybernetics at TUT, Akadeemia tee 21, 12618 Tallinn
Phone	6204240
E-mail	ando@cs.ioc.ee

Education

2004 – 2008	Tallinn University of Technology, Phd studies in computer science
2003 – 2004	Internship at INRIA, Sophia Antipolis (10.2003–01.2004)
2002 – 2004	Tallinn University of Technology, BSc studies in computer science
1998 – 2002	Tallinn University of Technology, MSc studies in computer science

Positions held

2005 – ...	Institute of Cybernetics at TUT; Researcher
2008 – 2008	Internship at Microsoft Research, Redmond (02.2008 – 04.2008)
2002 – 2005	Institute of Cybernetics at TUT; Engineer
1999 – 2001	Stockholm Environmental Institute Tallinn Centre (SEI-Tallinn); Programmer

Administrative responsibilities

2007	Testcom-FATES/FORTE 2007, Tallinn, 26-29 June 2007; member of the organizing committee
2006	MPC/AMAST 2006, Kuressaare, 2-8 July 2006; member of the organizing committee
2005	4th Estonian Summer School in Computer and System Science (ESSCaSS'05), Pedase, 7-12 August 2005; member of the organizing committee
2005	TCP/ICFP/GPCE 2005, Tallinn, 23 Sept.-1 Oct. 2005; member of the organizing committee
2004	2nd APPSEM II Workshop, Tallinn, 14-16 April 2004; member of the organizing committee
2004	3rd Estonian Summer School in Computer and System Science (ESSCaSS'04), Pedase, 8-12 August 2004; member of the organizing committee

Elulookirjeldus

Isikuandmed

Nimi	Ando Saabas
Sünniaeg	21.09.1979
Kodakondsus	Eesti

Kontaktandmed

Address	TTÜ Küberneetika Instituut, Akadeemia tee 21, 12618 Tallinn
Telefon	6204240
E-post	ando@cs.ioc.ee

Hariduskäik

2004 – 2008	Tallinna Tehnikaülikool, doktoriõpingud informaatikas
2003 – 2004	Intern Prantsuse Rahvuslikus Teadusinstituudis INRIA, Sophia Antipolises (10.2003–01.2004)
2002 – 2004	Tallinna Tehnikaülikool, magistriõpingud informaatikas
1998 – 2002	Tallinna Tehnikaülikool, bakalaureuseõpingud informaatikas

Teenistuskäik

2005 – ...	Tallinna Tehnikaülikool, Küberneetika Instituut; teadur
2008 – 2008	Intern Microsoft Researchi Redmondi uurimislaboris (02.2008 – 04.2008)
2002 – 2005	Tallinna Tehnikaülikool, Küberneetika Instituut; insener
1999 – 2001	Säästva Eesti Instituut (SEI-Tallinn); programmeerija

Teadusorganisatsiooniline ja -administratiivne tegevus

2007	Testcom-FATES/FORTE 2007, Tallinn, 26.-29. juuni 2007; korraldustoimkonna liige
2006	MPC/AMAST 2006, Kuressaare, 2.-8. juuli 2006; korraldustoimkonna liige
2005	4th Estonian Summer School in Computer and System Science (ESSCaSS'05), Pedase, 7.-12. august 2005; korraldustoimkonna liige
2005	TCP/ICFP/GPCE 2005, Tallinn, 23. sept. - 1. okt. 2005; korraldustoimkonna liige
2004	2nd APPSEM II Workshop, Tallinn, 14.-16. aprill 2004; korraldustoimkonna liige
2004	3rd Estonian Summer School in Computer and System Science (ESSCaSS'04), Pedase, 8.-12. august 2004; korraldustoimkonna liige